

Lecture Notes in Computer Science

1697

**Jack Dongarra Emilio Luque
Tomàs Margalef (Eds.)**

Recent Advances in Parallel Virtual Machine and Message Passing Interface

**6th European PVM/MPI Users' Group Meeting
Barcelona, Spain, September 1999
Proceedings**



Springer

Springer

Berlin

Heidelberg

New York

Barcelona

Hong Kong

London

Milan

Paris

Singapore

Tokyo

Jack Dongarra Emilio Luque
Tomàs Margalef (Eds.)

Recent Advances in Parallel Virtual Machine and Message Passing Interface

6th European PVM/MPI Users' Group Meeting
Barcelona, Spain, September 26-29, 1999
Proceedings



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Jack Dongarra
University of Tennessee and Oak Ridge National Laboratory
107, Ayres Hall, Knoxville, TN 37996-1301, USA
E-mail: dongarra@cs.utk.edu

Emilio Luque
Tomàs Margalef
Universitat Autònoma de Barcelona, Computer Science Department
Computer Architecture and Operating Systems Group
E-08193 Bellaterra, Barcelona, Spain
E-mail: {e.luque, t.margalef}@cc.uab.es

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Recent advances in parallel virtual machine and message passing interface :
proceedings / 6th European PVM-MPI Users' Group Meeting, Barcelona, Spain,
September 26 - 29, 1999. Jack Dongarra ... (ed.). - Berlin ; Heidelberg ; New
York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo :
Springer, 1999
(Lecture notes in computer science ; Vol. 1697)
ISBN 3-540-66549-8

CR Subject Classification (1998): D.1.3, D.3.2, F.1.2, G.1.0, B.2.1, C.1.2, C.2.4

ISSN 0302-9743

ISBN 3-540-66549-8 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

© Springer-Verlag Berlin Heidelberg 1999
Printed in Germany

Typesetting: Camera-ready by author

SPIN: 10704892 06/3142 —5 4 3 2 1 0

Printed on acid-free paper

Preface

Parallel Virtual Machine (PVM) and Message Passing Interface (MPI) are the most frequently used tools for programming according to the message passing paradigm, which is considered one of the best ways to develop parallel applications.

This volume comprises 67 revised contributions presented at the Sixth European PVM/MPI Users' Group Meeting, which was held in Barcelona, Spain, 26-29 September 1999. The conference was organized by the Computer Science Department of the Universitat Autònoma de Barcelona.

This conference has been previously held in Liverpool, UK (1998) and Cracow, Poland (1997). The first three conferences were devoted to PVM and were held at the TU Munich, Germany (1996), ENS Lyon, France (1995), and University of Rome (1994).

This conference has become a forum for users and developers of PVM, MPI, and other message passing environments. Interaction between those groups has proved to be very useful for developing new ideas in parallel computing and for applying some of those already existent to new practical fields.

The main topics of the meeting were evaluation and performance of PVM and MPI, extensions and improvements to PVM and MPI, algorithms using the message passing paradigm, and applications in science and engineering based on message passing. The conference included 3 tutorials on advances in PVM and MPI, 6 invited talks on various message passing issues such as advanced use of MPI and PVM and integration of tools and environments, and high performance clusters; and 33 oral presentations together with 34 poster presentations. These proceedings reflect the final results of the meeting.

This final version of the invited talks and presentations has been made possible by the kind contributions of the members of the PVM/MPI '99 program committee. Each paper submitted to the conference has been reviewed by at least two reviewers. We would like to congratulate the reviewers for their efforts.

This conference has received the kind support of the following companies and administrations: Ministerio de Educación y Ciencia, Generalitat de Catalunya, Universitat Autònoma de Barcelona, EuroTools consortium, SGI and Microsoft.

September 1999

Jack Dongarra
Emilio Luque
Tomàs Margalef

Program Committee

Vassil Alexandrov	University of Liverpool, UK
Ranieri Baraglia	CNUCE, Pisa, ITALY
Arndt Bode	LRR - Technische Universität München, GERMANY
Peter Brezany	University of Vienna, AUSTRIA
Marian Bubak	Institute of Computer Science - University of Mining and Metallurgy - Krakow, POLAND
Shirley Browne	University of Tennessee, USA
Jacques Chassin	IMAG —LMC - Grenoble, FRANCE
Jens Clausen	Technical University of Denmark, DENMARK
Jeremy Cook	Parallab - University of Bergen, NORWAY
Yiannis Cotronis	University of Athens, GREECE
José Cunha	Universidade Nova de Lisboa, PORTUGAL
Ivan Dimov	Bulgarian Academy of Science - Sofia, BULGARIA
Jack Dongarra	University of Tennessee and ORNL, USA
Graham Fagg	University of Tennessee, USA
Afonso Ferreira	INRIA - Sophia-Antipolis, FRANCE
Al Geist	Oak Ridge National Labs, USA
Michael Gerndt	Forschungszentrum Juelich, GERMANY
Rolf Hempel	C&C, Research Labs. - NEC Europe Ltd., GERMANY
Erik D'Hollander	University of Gent, BELGIUM
Ladislav Hluchy	Slovak Academy of Science - Bratislava, SLOVAKIA
Robert Hood	NASA Ames Research Center, USA
Peter Kacsuk	SZTAKI, HUNGARY
Henryk Krawczyk	Technical University of Gdansk, POLAND
Jan Kwiatkowski	Wroclaw University of Technology, POLAND
Miron Livny	University of Wisconsin —Madison, USA
Thomas Ludwig	LRR - Technische Universität München, GERMANY
Emilio Luque	Universitat Autònoma de Barcelona, SPAIN
Tomàs Margalef	Universitat Autònoma de Barcelona, SPAIN
Hermann Mierendorff	GMD, GERMANY
Barton Miller	University of Wisconsin —Madison, USA
Benno Overeinder	University of Amsterdam, THE NETHERLANDS
Andrew Rau-Chaplin	Dalhousie University - Halifax, CANADA
Yves Robert	Ecole Normale Supérieure de Lyon, FRANCE
Casiano Rodríguez	Universidad de La Laguna, SPAIN
Subhash Saini	NASA Ames Research Center, USA
Wolfgang Schreiner	RISC - University of Linz, AUSTRIA
Miquel A. Senar	Universitat Autònoma de Barcelona, SPAIN
Joao Gabriel Silva	Universidade de Coimbra, PORTUGAL
Vaidy Sunderam	Emory University - Atlanta, USA
Francisco Tirado	Universidad Complutense de Madrid, SPAIN
Bernard Tourancheau	Université Claude Bernard de Lyon, FRANCE

Pavel Tvrdik	Czech Technical Univeristy, CZECH REPUBLIC
Marian Vajtersic	Slovak Academy of Science - Bratislava, SLOVAKIA
Stephen Winter	University of Westminster, UK
Jerzy Wásniewski	The Danish Computing Centre for Research and Education, Lyngby, DENMARK
Roland Wismueller	LRR - Technische Universität München, GERMANY
Zahari Zlatev	National Environmental Research Institute - Copenhagen, DENMARK

Additional Reviewers

Jan Astalos	Institute of Informatics, SAS, Slovakia
Oliver Briant	IMAG- LMC, Grenoble, France
Nuno Correia	Universidade Nova de Lisboa, Portugal
K. De Bosschere	University of Ghent, Belgium
Francisco de Sande	Universidad de La Laguna, Spain
Miroslav Dobrucky	Institute of Informatics, SAS, Slovakia
Philipp Drum	Lehrstuhl für Rechnerorganisation, Germany
Vitor Duarte	Universidade Nova de Lisboa, Portugal
Milagros Fernández	Universidad Complutense de Madrid, Spain
R. Ferrini	CNUCE, Italy
Wlodzimierz Funika	Institute of Computer Science, AGH, Krakow, Poland
Jerome Galtier	INRIA, Sophia-Antipolis, France
Jesús A. González	Universidad de La Laguna, Spain
Owen Kaser	University of New Brunswick, Canada
D. Laforenza	CNUCE, Italy
Erwin Laure	Institute of Software Technology & Parallel Systems, University of Vienna, Austria
Coromoto León	Universidad de La Laguna, Spain
Markus Lindermeier	Technische Universität München, Germany
Ignacio M. Llorente	Universidad Complutense de Madrid, Spain
Róbert Lovas	SZTAKI, Hungary
Zsolt Nemeth	SZTAKI, Hungary
Karl L. Paap	GMD-SET, Germany
R. Perego	CNUCE, Italy
C. D. Pham	University of Lyon, France
Norbert Podhorszki	SZTAKI, Hungary
Paula Prata	Universidade da Beira Interior, Portugal
Günter Raki	Technische Universität München, Germany
José Luis Roda	Universidad de La Laguna, Spain
David Sagnol	INRIA- Sophia-Antipolis, France
Krzysztof Sowa	Institute for Software Technology & Parallel Systems, University of Vienna, Austria
T. Theoharis	University of Athens, Greece
Carsten Trinitis	Technische Universität München, Germany
Sipkova Viera	University of Vienna, Austria
Tran D. Viet	Institute of Informatics, SAS, Slovakia
J. M. Vincent	University of Grenoble, France
Roland Westrelin	University Claude Bernard, Lyon, France
Yijun Yu	University of Ghent, Belgium

Local Committee

Miquel A. Senar	Universitat Autònoma de Barcelona, Spain
José Antonio Marco	Universitat Autònoma de Barcelona, Spain
Bahjat Moh'd Qazzaz	Universitat Autònoma de Barcelona, Spain
Antonio Espinosa	Universitat Autònoma de Barcelona, Spain

Table of Contents

1 Evaluation and Performance

Performance Issues of Distributed MPI Applications in a German Gigabit Testbed	3
T. Eickermann, H. Grund, and J. Henrichs	
Reproducible Measurements of MPI Performance Characteristics	11
W. Gropp and E. Lusk	
Performance Evaluation of the MPI/MBCF with the NAS Parallel Benchmarks	19
K. Morimoto, T. Matsumoto, and K. Hiraki	
Performance and Predictability of MPI and BSP Programs on the CRAY T3E	27
J.A. González, C. Rodríguez, J.L. Roda, D.G. Morales, F. Sande, F. Almeida, and C. León	
Automatic Profiling of MPI Applications with Hardware Performance Counters	35
R. Rabenseifner	
Monitor Overhead Measurement with SKaMPI	43
D. Kranzlmüller, R. Reussner, and Ch. Schaubschläger	
A Standard Interface for Debugger Access to Message Queue Information in MPI	51
J. Cownie and W. Gropp	
Towards Portable Runtime Support for Irregular and Out-of-Core Computations	59
M. Bubak and P. Łuszczek	
Enhancing the Functionality of Performance Measurement Tools for Message Passing Environments	67
M. Bubak, W. Funika, K. Iskra, R. Maruszewski, and R. Wismüller	

Performance Modeling Based on PVM H. Mierendorff and H. Schwamborn	75
Efficient Replay of PVM Programs M. Neyman, M. Bukowski, and P. Kuzora	83
Relating the Execution Behaviour with the Structure of the Application A. Espinosa, F. Parcerisa, T. Margalef, and E. Luque	91
 2. Extensions and Improvements	
Extending PVM with Consistent Cut Capabilities: Application Aspects and Implementation Strategies A. Clematis and V. Gianuzzi	101
Flattening on the Fly: Efficient Handling of MPI Derived Datatypes J. L. Träff, R. Hempel, H. Ritzdorf, and F. Zimmermann	109
PVM Emulation in the Harness Metacomputing System: A Plug-In Based Approach M. Migliardi and V. Sunderam	117
Implementing MPI-2 Extended Collective Operations P. Silva and J. G. Silva	125
Modeling MPI Collective Communications on the AP3000 Multicomputer J. Touriño and R. Doallo	133
MPL*: Efficient Record/Replay of Nondeterministic Features of Message Passing Libraries J. Chassin de Kergommeaux, M. Ronsse, and K. De Bosschere	141
Comparison of PVM and MPI on SGI Multiprocessors in a High Bandwidth Multimedia Application R. Kutil and A. Uhl	149

On Line Visualization or Combining the Standard ORNL PVM with a Vendor PVM Implementation	157
J. Borkowski	
Native Versus Java Message Passing	165
N. Stankovic and K. Zhang	
JPT: A Java Parallelization Tool	173
K. Beyls, E. D'Hollander, and Y. Yu	
Facilitating Parallel Programming in PVM Using Condensed Graphs	181
J. P. Morrison and R. W. Connolly	
Nested Bulk Synchronous Parallel Computing	189
F. de Sande, C. León, C. Rodríguez, J. Roda, and J. A. González	
 3. Implementation Issues	
 An MPI Implementation on the Top of the Virtual Interface Architecture	199
M. Bertozzi, F. Boselli, G. Conte, and M. Reggiani	
MiMPI: A Multithread-Safe Implementation of MPI	207
F. García, A. Calderón, and J. Carretero	
Building MPI for Multi-Programming Systems Using Implicit Information	215
F. C. Wong, A.C. Arpaci-Dusseau, and D.E. Culler	
The Design for a High Performance MPI Implementation on the Myrinet Network	223
L. Prylli, B. Tourancheau, and R. Westrelin	
Implementing MPI's One-Sided Communications for WMPI	231
F. E. Mourão and J. G. Silva	

4. Tools

A Parallel Genetic Programming Tool Based on PVM	241
F. Fernández, J. M. Sánchez, M. Tomassini, and J.A. Gómez	
Net-Console: A Web-Based Development Environment for MPI Programs	249
A. Papagapiou, P. Evripidou, and G. Samaras	
Visual MPI, A knowledge-Based System for Writing Efficient MPI Applications	257
D. Ferenc, J. Nabrzyski, M. Stroiński, and P. Wierzejewski	

5. Algorithms

Solving Generalized Boundary Value Problems with Distributed Computing and Recursive Programming	267
I. Szeberényi and G. Domokos	
Hyper-Rectangle Distribution Algorithm for Parallel Multi-Dimensional Numerical Integration	275
R. Čiegis, R. Šablinskas, and J. Waśniewski	
Parallel Monte Carlo Algorithms for Sparse SLAE Using MPI	283
V. Alexandrov and A. Karaivanova	
A Method for Model Parameter Identification Using Parallel Genetic Algorithms	291
J. I. Hidalgo, M. Prieto, J. Lanchares, F. Tirado, B. de Andrés, S. Esteban, and D. Rivera	
Large-Scale FE Modelling in Geomechanics: A Case Study in Parallelization	299
R. Blaheta, O. Jakl, and J. Starý	
A Parallel Robust Multigrid Algorithm Based on Semi-Coarsening	307
M. Prieto, R. Santiago, I. M. Llorente, and F. Tirado	

6. Applications in Science and Engineering

PLIERS: A Parallel Information Retrieval System Using MPI	317
A. MacFarlane, J. A. McCann , and S.E. Robertson	
Parallel DSIR Text Retrieval System	325
A. Rungsawang, A. Tangpong , and P. Laohawee	
PVM Implementation of Heterogeneous ScaLAPACK Dense Linear Solvers	333
V. Boudet, F. Rastello, and Y. Robert	
Using PMD to Parallel Solve Large-Scale Navier-Stokes Equations. Performance Analysis on SGI/CRAY-T3E Machine	341
J. Chergui	
Implementation Issues of Computational Fluid Dynamics Algorithms on Parallel Computers	349
J. Płazek, K. Banaś, and J. Kitowski	
A Scalable Parallel Gauss-Seidel and Jacobi Solver for Animal Genetics	356
M. Larsen and P. Madsen	
Parallel Approaches to a Numerically Intensive Application Using PVM	364
R. Baraglia, R. Ferrini, D. Laforenza, and A. Laganà	
Solving the Inverse Toeplitz Eigenproblem Using ScaLAPACK and MPI	372
J. M. Badía and A. M. Vidal	
A Parallel Implementation of the Eigenproblem for Large, Symmetric and Sparse Matrices	380
E.M. Garzón and I. García	
Parallel Computation of the SVD of a Matrix Product	388
J. M. Claver, M. Mollar, and V. Hernández	
Porting Generalized Eigenvalue Software on Distributed Memory Machines Using Systolic Model Principles	396
P. Bassomo, I. Sakho, and A. Corbel	

Heading for an Asynchronous Parallel Ocean Model J. Schuele	404
Distributed Collision Handling for Particle-Based Simulation G. Frugoli, A. Fava, E. Fava, and G. Conte	410
Parallel Watershed Algorithm on Images from Cranial CT-Scans Using PVM and MPI on a Distributed Memory System C. Nicolescu, B. Albers, and P. Jonker	418
MPIPOV: A Parallel Implementation of POV-Ray Based on MPI A. Fava, M. Fava, and M. Bertozzi	426
Minimum Communication Cost Fractal Image Compression on PVM P. -Y. Wu	434
Cluster Computing Using MPI and Windows NT to Solve the Processing of Remotely Sensed Imagery J. A. Gallud, J. M. García, and J. García-Consuegra	442
Ground Water Flow Modelling in PVM L. Hluchý, V. D. Tran, L. Halada, and M. Dobrucký	450
 7. Networking	
Virtual BUS: A Simple Implementation of an Effortless Networking System Based on PVM S. Ishihara, S. Tani, and A. Takahara	461
Collective Communication on Dedicated Clusters of Workstations L. P. Huse	469
Experiences Deploying a Distributed Parallel Processing Environment over a Broadband Multiservice Network J. Corbacho-Lozano., O.-I. Lepe-Aldama., J. Solé-Pareta, and J. Domingo-Pascual	477

Asynchronous Communications in MPI – the BIP/Myrinet Approach	485
F. Chaussumier, F. Desprez, and L. Prylli	
Parallel Computing on PC Clusters – An Alternative to Supercomputers for Industrial Applications	493
M. Eberl, W. Karl, C. Trinitis, and A. Blaszczyk	
Benchmarking the PVM Group Communication Efficiency	499
M.R.Matuszek, A. Mazurkiewicz, and P. W. Umiński	
 8. Heterogeneous Distributed Systems	
Dynamic Assignment with Process Migration in Distributed Environments	509
P. Czarnul and H. Krawczyk	
Parallelizing of Sequential Annotated Programs in PVM Environment	517
A. Godlevsky, M. Gažák, and L. Hluchý	
Di_pSystem: A Parallel Programming System for Distributed Memory Architectures	525
F. Silva, H. Paulino, and L. Lopes	
Parallel NLP Strategies Using PVM on Heterogeneous Distributed Environments	533
G. E. Vazquez and N. B. Brignole	
Using PVM for Distributed Logic Minimization in a Network of Computers	541
L. Parrilla, J. Ortega, and A. Lloris	
Author Index	549

Performance Issues of Distributed MPI Applications in a German Gigabit Testbed

T. Eickermann¹, H. Grund², and J. Henrichs³

¹ Zentralinstitut für Angewandte Mathematik, Forschungszentrum Jülich,
D-52425 Jülich, Germany

² Institut für Algorithmen und Wissenschaftliches Rechnen,
GMD – Forschungszentrum Informationstechnik, Schloß Birlinghoven,
D-53754 Sankt Augustin, Germany

³ Pallas GmbH, Hermülheimer Str. 10, D-50321 Brühl, Germany

Abstract. The Gigabit Testbed West is a testbed for the planned upgrade of the German Scientific Network B-WiN. It is based on a 2.4 Gigabit/second ATM connection between the Research Centre Jülich and the GMD – National Research Center for Information Technology in Sankt Augustin. This contribution reports on those activities in the testbed that are related to metacomputing. It starts with a discussion of the IP connectivity of the supercomputers in the testbed. The achieved performance is compared with MetaMPI, an MPI library that is tuned for the use in metacomputing environments with high-speed networks. Applications using this library are briefly described.

1 Introduction

Traditionally, massively parallel (MPP) and vector-supercomputers are used as stand-alone machines. The availability of high-speed wide-area networks makes it feasible to distribute applications with high demands of CPU and/or memory over several such machines. This approach is followed by projects all over the world [1–3]. However, to make this ‘metacomputing’ approach successful, a couple of problems have to be solved. The first of them is to attach the supercomputers to the network with reasonable performance. This is a nontrivial task, since such machines are often not optimized for external communication. Secondly, there is a need for communication libraries like MPI that offer a high level of abstraction to the application programmer. In order to fully utilize especially a high-speed network some attention has to be paid to the overhead that is introduced by such a library. Finally, the applications have to take into account that the performance characteristics of the external network can’t compete with those of the MPP-internal communication.

A German project dealing with these issues is the ‘Gigabit Testbed West’. It started in August 1997 as a joint project of the Research Centre Jülich (FZJ) and the GMD – National Research Center for Information Technology in Sankt Augustin and is aimed to prepare the upgrade of the German Scientific Network B-WiN to Gigabit per second capacity which is scheduled for spring 2000. It is

funded by the DFN, the institution that operates the B-WiN. In the first year of operation the two research centers — which are approximately 120 km apart — were connected by an OC-12 ATM link (622 Mbit/s) based upon Synchronous Digital Hierarchy (SDH/STM4) technology. In August 1998 this link has been upgraded to OC-48 (2.4 Gbit/s).

Besides several institutes in the research centers in Jülich and Sankt Augustin other institutions participate in the testbed with their applications. These are the Alfred Wegener Institute for Polar and Marine Research (AWI), the German Climate Computing Center (DKRZ), the Universities of Cologne and Bonn, the National German Aerospace Research Center (DLR) in Cologne, the Academy of Media Arts in Cologne, and the industrial partners Pallas GmbH and echtzeit GmbH.

In this contribution, we report on our activities to set up a metacomputing environment with the parallel computers in Jülich and Sankt Augustin. We first describe the actions we took to improve the connectivity of those computers. Then we give some preliminary performance data of an MPI library, which is currently developed by Pallas for the use in the testbed. We end with a list of the applications that make use of this library, briefly discussing their communication needs.

2 Supercomputer Connectivity

Like in the 155 Mbit/s German Scientific Network, the backbone of the Gigabit Testbed West is an ATM network. In both research centers, a FORE Systems ATM switch (ASX-4000) connects the local networks to the OC-48 line. Since each router in a communication path results in an extra delay of up to a millisecond, the best solution with respect to performance would be to have all machines of the metacomputer in a single classical IP ATM network. Unfortunately, ATM connectivity for supercomputers has evolved quite slowly. While 622 Mbit/s interfaces are now available for all common workstation platforms, solutions are still outstanding for the major supercomputers used in the testbed. In Jülich, this is a Cray complex, consisting of two 512-node T3E MPPs and a 10-processor T90 vector-computer. For all of them, only 155 Mbit/s ATM interfaces are available (and will be in the foreseeable future). The same holds for a 35-node IBM SP2 in Sankt Augustin.

Therefore a different solution had to be found. The best-performing network connection of the Cray supercomputers is the 800 Mbit/s 'High Performance Parallel Interface' (HiPPI). About 430 Mbit/s have been achieved in the IP over HiPPI network of the Cray complex in Jülich. This is mainly due to the fact that HiPPI networks allow IP-packets of up to 64 KByte size (MTU size) and that a limiting factor for the performance is the rate at which the Crays can process IP packets.

In order to make use of that advantage for the testbed, it is essential that the router between the HiPPI and ATM network supports large MTUs on both media. In Jülich, dedicated workstations (currently an SGI O200 and a SUN

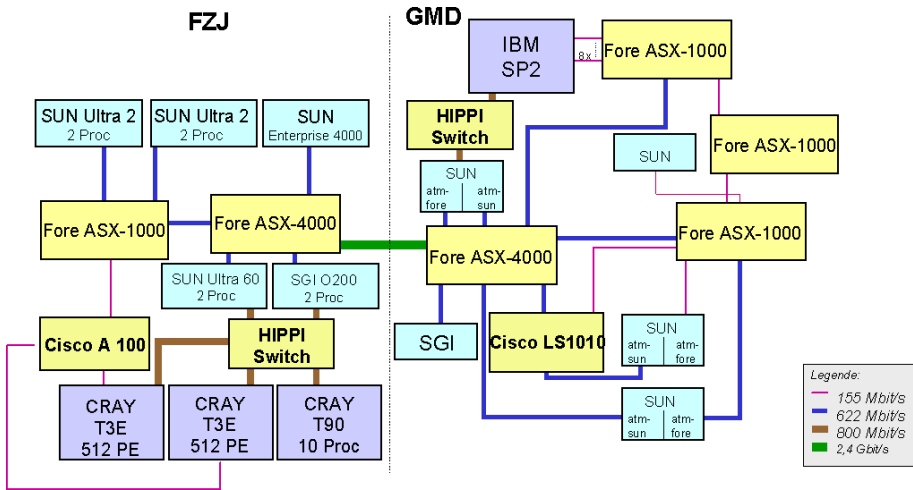


Fig. 1. Configuration of the Gigabit Testbed West in spring 1999. The FZJ in Jülich and the GMD in Sankt Augustin are connected via a 2.4 Gbit/s ATM link. The supercomputers are attached to the testbed via HiPPI-ATM gateways (and additionally via 155 Mbit/s ATM adapters), several workstations via 622 or 155 Mbit/s ATM interfaces.

Ultra 30) are used for that purpose. Both machines are equipped with a FORE Systems 622 Mbit/s ATM adapter supporting an MTU of 64 KByte.

A similar solution was chosen to connect the 35-node IBM SP2 in Sankt Augustin to the testbed. 8 SP2-nodes are equipped with 155 Mbit/s ATM adapters and one with a HiPPI interface. The ATM adapters are connected to the testbed via a FORE ASX 1000. The HiPPI network is routed by an 8-processor SUN E5000 which has also a FORE 622 Mbit/s ATM adapter. A 12-processor SGI Onyx 2 visualization server that is also used by applications in the testbed is equipped with a 622 Mbit/s ATM-interface. Figure 1 shows the current configuration of the testbed as described above.

Depending on the application, latency can be as important as bandwidth. An unavoidable delay is introduced by the travelling time of the signal in the wide-area network. In our case, this delay is about 0.9 msec for the 120 km SDH/ATM line. Each ATM switch adds about 10-20 μ sec. As our measurements show, a substantial delay can be introduced by the attached computers. Table 1 lists the results of delay/throughput measurements between various machines in the testbed. More details are given elsewhere [4]. The SUN Ultra 60 – SUN E5000 numbers show that the workstations and servers which are used as HiPPI-ATM gateways can make good use of the available bandwidth without adding a significant delay. The situation is different for the supercomputers. Even in the local HiPPI network in Jülich, a latency of 3 msec is measured on the application level. In order to estimate the effect of the HiPPI-ATM gateways the T3E – T3E measurements were repeated, routing the data the following way: T3E via HiPPI to Ultra 60 via ATM to O200 via HiPPI to the second T3E. This

Table 1. Throughput and delay measurements with a ping-pong program using TCP-stream sockets. The delay is half of the round-trip time of a 1 Byte message. For throughput measurements, messages of up to 10 MByte size were used. The bandwidth available for a TCP connection over the given network is listed as 'theoretical bandwidth'.

	network	theoretical bandwidth [MByte/s]	measured throughput [MByte/s]	untuned throughput [MByte/s]	delay [msec]
T3E — T3E	HiPPI	95.3	44.7	10.2	3.3
	HiPPI/ATM 622	67.3	34.4	6.4	4.0
	Ethernet	1.2	0.97	0.97	4.3
Ultra 60 – E5000	ATM 622	67.3	56.9	33.1	0.91
T3E — SP2	HiPPI/ATM 622	67.3	32.4	1.95	3.1
	ATM 155	16.8	12.5	6.5	2.6

adds only a delay of about 0.7 msec but substantially reduces the throughput. The performance of the T3E – SP2 connection is mainly limited by the I/O capabilities of the MicroChannel based SP2 nodes. For all measurements, large TCP buffers and TCP windows were used. The comparison with untuned sockets (using the operating system defaults) show the importance of socket tuning.

3 MPI Communication in the testbed

Many programs written for parallel computers use MPI [5], which offers a portable and efficient API. Therefore, to make metacomputing usable for a broader range of users, the availability of a metacomputing-aware MPI implementation is a must. With metacomputing-aware we mean that the communication both inside and between the machines that form the metacomputer should be efficient. Furthermore, a couple of features that are useful for metacomputing applications are part of the MPI-2 [6] definition. Dynamic process creation and attachment e.g. can be used for realtime-visualization or computational steering; language-interoperability is needed to couple applications that are implemented in different programming languages. When the project started, no MPI implementation with this set of features was available (to our knowledge, this is still true today). Therefore the development of such a library, named 'MetaMPI' was assigned to Pallas GmbH. The implementation is based on MPICH and uses different devices for the communication inside and between the coupled machines. The communication between the machines is handled by so-called router-PEs (processing elements). A message that has to travel from one machine to another is at first handed over to a local router-PE. From there it is transferred to a router-PE on the target machine using a TCP-socket connection. Finally this router-PE sends the message to the target PE. The current status of this development is

Table 2. Throughput and delay measurements with a ping-pong program using MetaMPI. The 'native' values refer to communication inside the machines.

network		measured throughput [MByte/s]	delay [msec]
T3E	native	333	0.030
SP2	native	42	0.14
T3E — T3E	HiPPI	35.2	4.8
	HiPPI/ATM 622	28.7	5.5
	Ethernet	0.96	5.8
T3E — SP2	HiPPI/ATM 622	16.5	5.1
	ATM 155	7.7	4.3

that a full MPI-1.2 library which supports the Cray T3E, the IBM SP2 and Solaris 2 machines is available. Tuning of the external communication and incorporation of the selected MPI-2 features is under way. A detailed description of the MetaMPI implementation is given in [7]. Here we focus on point-to-point communication performance and compare it with the TCP/IP measurements of the previous section.

A comparison of the delays shown in table 1 and 2 shows that MetaMPI introduces an additional delay of about 1.5 msec for external communication. The bandwidth that is available for external messages is limited by the fact that this message has to take three hops to reach its destination: from the sender to local router-PE, from there to a remote router-PE, and finally to the target. Assuming that the time needed for each hop is determined by the available bandwidth on that path leads to a net bandwidth that is good agreement with our measurements. This means that, whereas careful tuning of the MetaMPI implementation might further reduce the delay, the throughput is near optimal. It should be noted that the measurements used the **MPI_DOUBLE** datatype, which can be exchanged between the T3E and the SP2 without any conversion. Such conversions are necessary e.g. for integer datatypes and significantly reduce the available bandwidth.

4 Distributed MPI Applications

With MetaMPI, the structure of the metacomputer is completely transparent to the application. Nevertheless, it is generally not a good idea to run a tightly coupled parallel application on a metacomputer without taking care of the hierarchy of bandwidths and latencies. Increased communication time often results in reduced efficiency. A class of applications that do not suffer from this problem are so called 'coupled fields' simulations. Here, two or more space- and time-dependent fields interact with each other. When the fields are distributed

over the machines of the metacomputer these components are often only loosely coupled – leading to moderate communication requirements. A second class of applications benefits from being distributed over supercomputers of different architecture, because they contain partial problems which can best be solved on massively parallel or vector-supercomputers. For other applications, real-time requirements are the reason to connect several machines. We now briefly describe the MPI based applications that currently use the Gigabit Testbed West, with a focus on their communication patterns. Details on these applications will be given in separate publications.

1. Solute Transport in Ground Water

Partners: Institute for Petroleum and Organic Geochemistry, FZJ.

Synopsis: Two independent programs for ground water flow simulation (TRACE, FORTRAN 90) and transport of particles in a given water flow (PARTRACE, C++) are coupled. The distributed version will allow for larger simulations, since both applications have high CPU and memory requirements.

Communication: The 3-D water flow field is transferred from the IBM SP2 (TRACE) to the Cray T3E (PARTRACE) at the beginning of every timestep. For typical parameters, 10–100 MByte are transferred in a burst every 2–10 seconds.

2. MEG Analysis

Partners: Institute of Medicine, FZJ.

Synopsis: A parallel program (pmusic) that estimates the position and strength of current dipoles in a human brain from magnetoencephalography measurements using the MUSIC algorithm is distributed over a massively parallel and a vector supercomputer. One part of the MUSIC algorithm can use up to 100 CPUs very efficiently, while another needs a single fast CPU. Therefore superlinear speedup is expected when the application is run on a heterogeneous metacomputer.

Communication: Only few data are transferred, but the algorithm is sensitive to latency, because one cycle takes only a few milliseconds.

3. Distributed Climate and Weather Models

Partners: Alfred Wegener Institute for Polar and Marine Research, German Climate Computing Center, and the Institute for Algorithms and Scientific Computing (SCAI), GMD.

Synopsis: A parallel ocean-ice model (based on MOM-2) running on Cray T3E and a parallel atmospheric model (IFS) running on IBM SP2 are coupled with the CSM flux coupler that is also run on the T3E. Both programs have high CPU requirements.

Communication: Exchange of 2-D surface data in every timestep (typically 2 seconds), up to 1 MByte in short bursts. Although the average network load is small, high bandwidth is needed since the application blocks until all data are exchanged.

4. Distributed Fluid-Structure Interaction

Partners: Pallas GmbH and SCAI, GMD.

Synopsis: An open interface (COCOLIB) that allows the coupling of industrial structural mechanics and fluid dynamics codes has been developed in the EC-funded project CIPAR. This is ported to the metacomputing environment.

Communication: Depends on the coupled applications.

5. Multiscale Molecular Dynamics

Partners: Institute for Applied Mathematics, University of Bonn.

Synopsis: Two existing parallel programs that simulate molecular dynamics with short-range (MolGrid) and long-range (TreeMol) interactions will be coupled and shall run distributed on the Cray T3E in Jülich and the PARNASS parallel computer in Bonn.

Communication: Atom positions have to be exchanged in every timestep of the long-range/short-range interaction, resulting in an average load of 40 MByte/s. Latency is hidden by the algorithm.

6. Coupled Litospheric Processes

Partners: Institute for Applied Mathematics and Institute of Geodynamics, University of Bonn.

Synopsis: Coupled litospheric processes involving fluid dynamics, structural mechanics, heat exchange and chemical processes will be simulated by four parallel programs running on the Cray T3E, the IBM SP2 and PARNASS.

Communication: About 2 MByte (after an application-specific compression) must be transferred at each timestep. Such a timestep is expected to take 0.1 seconds.

5 Related Activities

The applications in cooperation with the University in Bonn rely on an extension of the testbed to Bonn that will be operable in spring 1999. Further extensions to the University, the National German Aerospace Research Center, and the Academy of Media Arts in Cologne are currently being installed and will be used by new application projects from the areas of multimedia and metacomputing. Another metacomputing application in the testbed deals with realtime analysis and visualization of functional magnetic resonance imaging (fMRI). It does not use MPI for the external communication and is described in more detail elsewhere [9].

6 Conclusions

In this contribution, we presented first results of our efforts to establish efficient communication for MPI based metacomputing applications in the Gigabit Testbed West. The underlying 2.4 Gbit/s SDH and ATM technology for the wide-area backbone is mature, a necessary condition for the upgrade of the

German Scientific Network that is planned for early 2000. The implementation of the HiPPI-ATM gateways for the Cray T3E and IBM SP2 resulted in a significant enhancement of their networking capabilities. Still there seems to be room for improvements in this area. The MetaMPI library imposes only little overhead to the raw TCP/IP communication (in terms of bandwidth), yet delivering a standard API for message passing programs. Nevertheless, there is a dramatic difference between the performance of the MPP-internal and the external communication. The applications of 'coupled fields' and 'heterogenous metacomputing' type will have to prove that this difference can be compensated for.

Acknowledgements

Many of the activities that are reported in this contribution are not the work of the authors but of several persons in the institutions that participate in the Gigabit Testbed West project. The authors wish to thank R. Niederberger, M. Sczimarowsky, from the Research Centre Jülich, U. Eisenblätter, F. Hommes, P. Wunderling, and L. Zier at the GMD, J. Worringen, M. Pöppe, T. Bemmerl RWTH Aachen, to mention but a few. We also wish to thank the BMBF for partially funding the Gigabit Testbed West and the DFN for its support.

References

1. I. Foster and C. Kesselman, The Globus Project: A Status Report. Proc. IPPS/SPDP '98 Heterogeneous Computing Workshop, pp 4-18, 1998.
2. H.E. Bal, A. Plaat, T. Kielmann, J. Maassen, R. van Nieuwpoort, and R. Veldema, Parallel Computing on Wide-Area Clusters: the Albatros Project, Proc. Extreme Linux Workshop, pp. 20-24, Monterey, CA, June 8-10, 1999.
3. E. Gabriel, M. Resch, T. Beisel, R. Keller, Distributed computing in a heterogeneous computing environment, in V. Alexandrov and J. Dongarra (eds.) Recent Advances in PVM and MPI, pp 180-197, Springer 1998.
4. H. Grund, F. Hommes, R. Niederberger, and E. Pless, High Speed Supercomputer Communications in Broadband Networks, TERENA-NORDUnet Networking Conference, Sweden, June 1999.
5. Message Passing Interface Forum, MPI: A Message-Passing Interface Standard, University of Tennessee, <http://www.mcs.anl.gov/mpi/index.html>, 1995.
6. Message Passing Interface Forum, MPI-2: Extensions to the Message-Passing Interface, University of Tennessee, <http://www.mcs.anl.gov/mpi/index.html>, 1997.
7. J. Worringen, M. Pöppe, T. Bemmerl, J. Henrichs, T. Eickermann, and H. Grund, MetaMPI – an extension of MPICH for Metacomputing Environments, submitted to ParCo99, Delft, August 1999.
8. G.D. Burns, R.B. Daoud, and J.R. Vaigl, LAM: An Open Cluster Environment for MPI, Supercomputing Symposium '94, Toronto, Canada, June 1994.
9. T. Eickermann, W. Frings, S. Posse, and R. Völpe, Distributed Applications in a German Gigabit WAN, accepted for High Performance Distributed Computing, Los Angeles, August 1999.

Reproducible Measurements of MPI Performance Characteristics*

William Gropp and Ewing Lusk

Argonne National Laboratory, Argonne, IL, USA

Abstract. In this paper we describe the difficulties inherent in making accurate, reproducible measurements of message-passing performance. We describe some of the mistakes often made in attempting such measurements and the consequences of such mistakes. We describe `mpptest`, a suite of performance measurement programs developed at Argonne National Laboratory, that attempts to avoid such mistakes and obtain reproducible measures of MPI performance that can be useful to both MPI implementors and MPI application writers. We include a number of illustrative examples of its use.

1 Introduction

Everyone wants to measure the performance of their systems, but different groups have different reasons for doing so:

- Application writers need understanding of the performance profiles of MPI implementations in order to choose effective algorithms for target computing environments.
- Evaluators find performance information critical when deciding which machine to acquire for use by their applications.
- Implementors need to be able to understand the behavior of their own MPI implementations in order to plan improvements and measure the effects of improvements made.

All of these communities share a common requirement of their tests: that they be *reproducible*. As obvious as this requirement is, it is difficult to satisfy in practice. Parallelism introduces an element of nondeterminism that must be tightly controlled. The Unix operating system, together with network hardware and software, also introduces sporadic intrusions into the test environment that must be taken into account. The very portability of MPI suggests that the performance of the same operations (MPI function calls) can be meaningfully compared among various parallel machines, even when the calls are implemented in quite different ways. In this paper we review the perils of shortcuts frequently taken in

* This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

attempting to acquire reproducible results, and the approach we take to avoid such perils.

Over the years, the MPICH group has developed a suite of programs that characterize the performance of a message-passing environment. An example is the program `mpptest` that can be used to quickly characterize the performance of an MPI implementation in a variety of ways. For example, it was used to quickly measure the performance of a variety of pre-MPI message-passing implementations and to identify message sizes where sharp performance transitions occurred; see [4], Figures 1 and 3. An example of the use of `mpptest` in tuning an MPI implementation is shown in [3]. These programs are portable to any MPI implementation.

A number of parallel performance tests already exist. Many of these strive to be benchmarks that produce a “figure of merit” for the system. Our interest is in the details of the behavior, rather than a few numbers representing the system. Perhaps the closest project to our work is SKaMPI [7]. The SKaMPI system provides graphical output detailing the behavior of a wide variety of MPI functions and includes an adaptive message-length refinement algorithm similar to that in `mpptest`. The testing method in SkaMPI is somewhat different from ours and uses different rules when accepting experiments.

A number of well-known benchmarks are accessible through BenchWeb [1]. The ParkBench Organization provides a variety of codes, including “compact application benchmarks”. A review of some of the issues in developing a benchmark may be found in [2]. Previous PVMMPI meetings have included papers on performance measurement; see, for example, [5, 6].

We hope that the unusual approach taken in `mpptest` makes it a useful addition to the collection of performance measurement procedures for MPI programs. The paper first reviews (Section 2) some of the difficulties of performance performance measurements and characterizations. Section 3 briefly describes the testing methods and variations, relating the choices to the issues raised in Section 2. Section 4 presents a few examples that illustrate the capabilities of our performance characterization tests.

2 Perils of Performance Measurement

Simple tests can be misleading. As communication software becomes more sophisticated (for example, dynamically allocating resources to busy communication channels), simple tests become less indicative of the performance of a full application. The following list describes some of the pitfalls in measuring communication performance, in the form of “mistakes often made.”

1. **Forget to establish initial communication link.** Some systems dynamically create connections. The first communication between two processes can take far longer than subsequent communications.
2. **Ignore contention with unrelated applications or jobs.** A background file system backup may consume much of the available communication bandwidth.

3. **Ignore nonblocking calls.** High-performance kernels often involve non-blocking operations both for the possibility of communication overlap but, more important, for the advantage in allowing the system to schedule communications when many processes are communicating. Nonblocking calls are also important for correct operation of many applications.
4. **Ignore overlap of computation and communication.** High-performance kernels often strive to do this for the advantages both in data transfer and in latency hiding.
5. **Confuse total bandwidth with point-to-point bandwidth.** Dedicated, switched networks have very different performance than shared network fabrics.
6. **Make an apples-to-oranges comparison.** Message-passing accomplishes *two* effects: the transfer of data and a handshake (synchronization) to indicate that the data are available. Some comparisons of message passing with remote-memory or shared-memory operations ignore the synchronization step.
7. **Compare CPU time to elapsed time.** CPU time may not include any time that was spent waiting for data to arrive. Knowing the CPU load caused by a message-passing system is useful information, but only the elapsed time may be used to measure the time it takes to deliver a message.
8. **Ignore correctness.** Systems that fail for long messages may have an unfair advantage for short messages.
9. **Time events that are small relative to the resolution of the clock.** Many timers are not cycle counters; timing a single event may lead to wildly inaccurate times if the resolution of the clock is close to the time the operation takes. A related error is to try to correct the clock overhead by subtracting an estimate of the time to call the clock that is computed by taking the average of the time it takes to call the clock; this will reduce the apparent time and artificially inflate performance.
10. **Ignore cache effects.** Does the data end up in the cache of the receiver? What if data doesn't start in the cache of the sender? Does the transfer of data perturb (e.g., invalidate) the cache?
11. **Use a communication pattern different from the application.** Ensuring that a receive is issued before the matching send can make a significant difference in the performance. Multiple messages between different processes can also affect performance. Measuring ping-pong messages when the application sends head-to-head (as many scientific applications do) can also be misleading.
12. **Measure with just two processors.** Some systems may poll on the number of possible sources of messages; this can lead to a significant degradation in performance for real configurations.
13. **Measure with a single communication pattern.** No system with a large number of processors provides a perfect interconnect. The pattern you want may incur contention. One major system suffers slowdowns when simple butterfly patterns are used.

The programs described in this paper attempt to avoid these problems; for each case, we indicate below how we avoid the related problem.

3 Test Methodology

In this section we discuss some of the details of the testing. These are related to the issues in measuring performance described in Section 2. Our basic assumption is that in any short measurement, the observed time will be perturbed by some positive time Δt and that the distribution of these perturbations is random with an unknown distribution. (There is one possible negative perturbation caused by the finite resolution of the clock; this is addressed by taking times much longer than the clock resolution.)

3.1 Measuring Time

The fundamental rule of testing is that the test should be repeatable; that is, running the test several times should give, within experimental error, the same result. It is well known that running the same program can produce very different results each time it is run.

The only time that is reproducible is the minimum time of a number of tests. This is the choice that we make in our testing. By making a number of tests, we eliminate any misleading results due to initialization of links (issue 1).

Using the minimum is not a perfect choice; in particular, users will see some sort of average time rather than a minimum time. For this reason, we provide an option to record the maximum time observed, as well as the average of the observations. While these values are not reproducible, they are valuable indicators of the variation in the measurements.

The next question concerns what the minimums should be taken over. We use a loop with enough iterations to make the cost of any timer calls inconsequential by comparison. This eliminates errors related to the clock resolution (issue 9). In addition, the length of time that the loop runs can be set; this allows the user to determine the tradeoff between the runtime (cost) of the test and its accuracy.

3.2 Message Lengths

The performance of data movement, whether for message-passing or simple memory copies, is not a simple function of length. Instead, the performance is likely to consist of a number of sudden steps as various thresholds are crossed. Sampling at regular or prespecified data lengths can give misleading results. The `mpptest` program can automatically choose message lengths. The rule that `mpptest` uses is to attempt to eliminate artifacts in a graph of the output. It does this by computing three times: $f(n_0)$, $f(n_1)$, and $f((n_0 + n_1)/2)$, where $f(n)$ is the time to send n bytes. Then `mpptest` estimates the error in interpolating between n_0 and n_1 with a straight line by computing the difference between $(f(n_0) + f(n_1))/2$ and $f((n_0 + n_1)/2)$. If this value is larger than a specified threshold (in relative

terms), the interval $[n_0, n_1]$ is subdivided into two intervals, and the step is repeated. This can continue until a minimum separation between message lengths is reached.

3.3 Scheduling of Tests

The events that cause perturbations in the timing of a program can last many milliseconds or more. Thus, a simple approach that looks for the minimum of averages within a short span of time can be confused by a single, long-running event. As a result, it is important to spread the tests for each message length over the full time of the characterization run. A sketch of the appropriate measurement loop is shown below:

```

for (number of repetitions) {
  for (message lengths) {
    Measure time for this length
    if this is the fastest time yet, accept it
  }
}

```

Note that this approach has the drawback that it doesn't produce a steady stream of results; only at the very end of the test are final results available. However, it comes much closer to what careful researchers already do—run the test several times and take the best results. This helps address contention with other applications or jobs (issue 2), though does not solve this problem.

Tests with anomalously high times, relative to surrounding tests, are automatically rerun to determine if those times reflect a property of the communication system or are the result of a momentary load on the system. This also aids in producing reproducible results.

Note also that it is important to run a number of cycles of this loop before refining the message intervals. Otherwise, noise in the measurements can generate unneeded refinements.

3.4 Test Operations

Rather than test only a single operation, such as the usual “ping pong” or round-trip pattern, our tests provide for a wide variety of tests, selected at run time through command line arguments. The following list summarizes the available options and relates them to the issues in Section 2.

Number of processors. Any number of processors can be used. By default, only two will communicate (this tests for scalability in the message-passing implementation itself; see issue 12). With the `-bisect` option, half of the processors send to the other half, allowing measurement of the bisection bandwidth of the system (issue 5).

Cache effects. The communication tests may be run using successive bytes in a large buffer. By selecting the buffer size to be larger than the cache size, all communication takes place in memory that is not in cache (issue 10).

Communication Patterns. A variety of communication patterns can be specified for the bisection case, addressing issue 13. In addition, both “ping pong” and head-to-head communication is available when testing two communicating processes. More are needed, in particular to make it easier to simulate an application’s communication pattern (issue 11).

Correctness. Correctness is tested by a separate program, **stress**. This program sends a variety of bit patterns in messages of various sizes, and checks each bit in the receive message. Running this test, along with the performance characterization tests for large message sizes, addresses issue 8.

Communication and computation overlap. A simple test using a fixed message length and a variable amount of computation provides a simple measurement of communication/computation overlap, addressing issue 4.

Nonblocking Communication. Nonblocking communication is important; in many applications, using nonblocking communication routines is the easiest way to ensure correctness by avoiding problems related to finite buffering. The performance of nonblocking routines can be different from the that of the blocking routines. Our tests include versions for nonblocking routines (issue 3).

4 Examples

This section shows the capabilities of the **mpptest** characterization program in the context of specific examples.

Discontinuous Behavior The need for adaptive message length choice can be seen in Figure 1(a). This illustrates why the simple latency and bandwidth model is inappropriate as a measure of performance of a system.

We see the stair-steps illustrating message packet sizes (128 bytes). We also see the characteristic change in protocol for longer messages. Here it looks like the protocol changes at 1024 bytes, and that it is too late. The implementation is not making an optimal decision for the message length at which to switch methods; slightly better performance could be achieved in the range of 768 to 1024 bytes by using the same method used for longer messages.

Figure 1(c) shows the behavior for nonblocking sends instead of blocking sends. Note the small but measurable overhead compared with Figure 1(a).

Cache Performance Effects Performance tests often use the same, relatively small, data arrays as the source and destination of messages. In many applications, data is moved from main memory, not from cache. Figure 1(b) shows an example where the performance with data in cache is better than when the data is not in cache, both in absolute terms and in the trend (lower slope for in-cache data).

Variation in Performance In an application, the minimum times for an operation may not be as important as the average or maximum times. Figure 1(d) shows

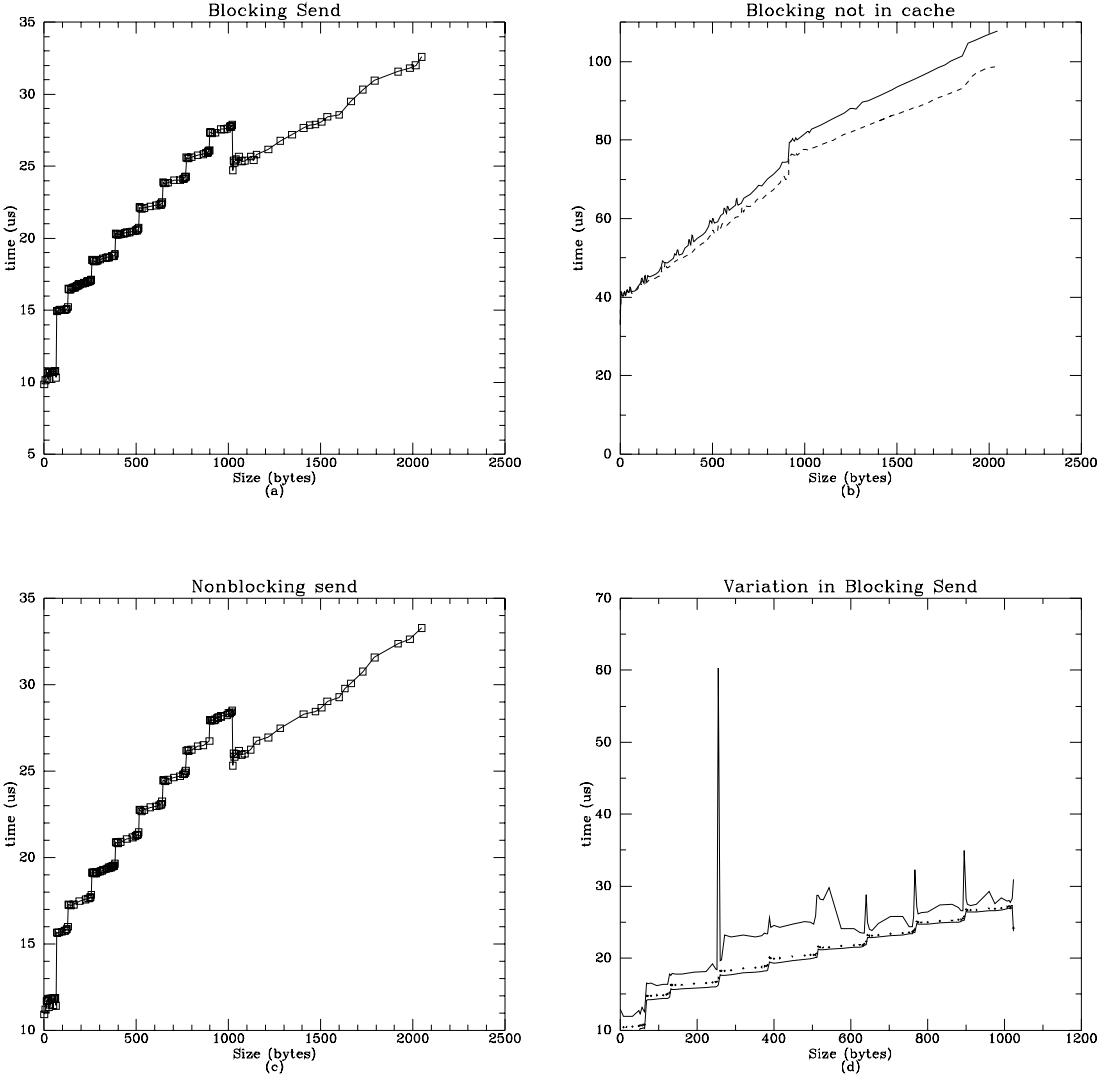


Fig. 1. Example results generated by `mpptest` on several platforms. Graph (a) shows an example of discontinuous performance identified by `mpptest`. Note the sharp drop in latency for zero bytes, the significant steps up to 1024 bytes, and the lower slope beyond 1024 bytes. Graph (b) shows an example of the change in performance between data in cache (dashed line) and not in cache (solid line). Graph (c) shows the extra cost of using nonblocking sends. Graph (d) gives an example of the variation in performance even on a system communicating with shared-memory hardware (see text for details).

the variations in times on a lightly-loaded shared-memory system. This graph is very interesting in that the minimum (bottom line) and average times (dots) are close together in most places, but the maximum observed time (top line) can be quite large. The peaks seem to line up with the transitions; however, since there are more measurements near the transitions, the correlation may be an accident. Further testing, particularly with the evenly spaced message sizes, would be required to determine if those peaks occurred only at the transitions.

5 Conclusion

We have illustrated the difficulty in characterizing performance and have discussed how the MPICH performance characterization programs can be used to discover properties of the parallel environment. The software is freely available from <http://www.mcs.anl.gov/mpi/mpich/perftest> or <ftp://ftp.mcs.anl.gov/pub/mpi/misc/perftest.tar.gz>.

References

1. Benchweb. World Wide Web. <http://www.netlib.org/benchweb/>.
2. Parkbench Committee. Public international benchmarks for parallel computers. *Scientific Programming*, 3(2):101–146, 1994. Report 1.
3. W. Gropp and E. Lusk. A high-performance MPI implementation on a shared-memory vector supercomputer. *Parallel Computing*, 22(11):1513–1526, January 1997.
4. W. D. Gropp and E. Lusk. Experiences with the IBM SP1. *IBM Systems Journal*, 34(2):249–262, 1995.
5. J. Piernas, A. Flores, and J. M. García. Analyzing the performance of MPI in a cluster of workstations based on fast Ethernet. In Marian Bubak, Jack Dongarra, and Jerzy Waśniewski, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface*, volume 1332 of *Lecture Notes in Computer Science*, pages 17–24. Springer, 1997. 4th European PVM/MPI Users' Group Meeting.
6. Michael Resch, Holger Berger, and Thomas Boenisch. A comparison of MPI performance on different MPPs. In Marian Bubak, Jack Dongarra, and Jerzy Waśniewski, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface*, volume 1332 of *Lecture Notes in Computer Science*, pages 25–32. Springer, 1997. 4th European PVM/MPI Users' Group Meeting.
7. R. Reussner, P. Sanders, L. Prechelt, and M. Müller. SKaMPI: A detailed, accurate MPI benchmark. In Vassuk Alexandrov and Jack Dongarra, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 52–59. Springer, 1998. 5th European PVM/MPI Users' Group Meeting.

Performance Evaluation of MPI/MBCF with the NAS Parallel Benchmarks

Kenji Morimoto, Takashi Matsumoto, and Kei Hiraki

Department of Information Science, Faculty of Science, University of Tokyo
7-3-1 Hongo, Bunkyo Ward, Tokyo 113-0033, Japan
{morimoto,tm,hiraki}@is.s.u-tokyo.ac.jp

Abstract. MPI/MBCF is a high-performance MPI library targeting a cluster of workstations connected by a commodity network. It is implemented with the Memory-Based Communication Facilities (MBCF), which provides software mechanisms for users to access remote task's memory space with off-the-shelf network hardware. MPI/MBCF uses *Memory-Based FIFO* for message buffering and *Remote Write* for communication without buffering from among the functions of MBCF. In this paper, we evaluate the performance of MPI/MBCF on a cluster of workstations with the NAS Parallel Benchmarks. We verify whether a message passing library implemented on the shared memory model achieves higher performance than that on the message passing model.

1 Introduction

The message passing model is a programming paradigm for a parallel processing environment. The Message Passing Interface (MPI) [3] is an interface-level instance of it. The shared memory model is another programming paradigm. When these two models are considered as communication models, they are exchangeable; one can emulate the other. In regard to implementation, a shared memory communication mechanism has advantages in performance over a message passing communication mechanism. This is because the former can transfer data directly to the destination memory while the latter needs intermediate buffering space. With the help of memory-oriented architectural supports such as MMU, the mechanism of shared memory communication can be implemented with lower software overheads than that of message passing communication even on packet-based off-the-shelf network hardware.

From the above consideration, we hold that message passing communication implemented with a high-performance shared memory communication mechanism gives better performance than that implemented with a message passing communication mechanism. We verify our claim by instrumentation of practical parallel applications in this paper. We chose MPI Ver. 1.2 [3,4] for a message passing communication library to be implemented. We used the Memory-Based Communication Facilities (MBCF) [7,8] as a software-implemented shared memory communication mechanism. The round-trip time and the peak bandwidth of our MPI library, called MPI/MBCF, show that the MBCF functions for shared

memory communication work effectively for the message passing library [10]. We employed the NAS Parallel Benchmarks [1,2] as benchmark applications to evaluate the behavior of MPI/MBCF in various communication patterns.

The rest of this paper is organized as follows. Section 2 gives the explanation of MBCF and MPI/MBCF, followed by the basic performance parameters of MPI/MBCF. We show the results of performance evaluation of MPI/MBCF with the NAS Parallel Benchmarks in Sect. 3 We conclude in Sect. 4 with a summary.

2 MPI/MBCF

2.1 Implementation of MPI/MBCF

MBCF is a software-based mechanism for accessing a remote memory space even on packet-based off-the-shelf network hardware such as Ethernet. We used *Remote Write* and *Memory-Based FIFO* for our MPI implementation from among the functions of MBCF. *Remote Write* enables a user to write data directly to remote task's logical address space. With *Memory-Based FIFO*, a user can send data to a remote FIFO-queue. The buffering area of the queue is reserved in the remote user's address space.

MPI/MBCF [10] is a complete implementation of MPI-1.2. To implement two fundamental point-to-point communication functions, send and receive, we employed two protocols for actual communication by the library. One is the *eager* protocol in terms of MPICH [5]. The other is the *write* protocol in our terms [9]. These two protocols are described with the behavior of the sender and the receiver as follows.

Eager Protocol. The sender sends the message header and data to the receiver with a pre-fixed address. The receiver examines matching of the message and pending receives to take out the data.

Write Protocol. The receiver sends the header to the source process. This header contains the address of the receive buffer. The sender examines matching of the header and pending sends to send the data by a remote write primitive.

The eager protocol enables the sender to send data as soon as the data gets ready. The write protocol enables the receiver to receive data without buffering. We combined these two protocols as follows.

- The receiver sends the header to the source process as a *request for sending* (based on the write protocol), if a matching message has not arrived yet and if the source process is uniquely specified.
- The sender sends the message data to the receiver by a remote write primitive (based on the write protocol) if a matching request has arrived. Or else the sender sends the message header and data to the receiver with a pre-fixed address (based on the eager protocol).

2.2 Basic Performance Parameters of MPI/MBCF

We measured the round-trip time and the peak bandwidth of MPI/MBCF on a cluster of workstations connected with a 100BASE-TX Ethernet. The following machines were used for measurement: Sun Microsystems SPARCstation 20 (85 MHz SuperSPARC \times 1), Sun Microsystems Fast Ethernet SBus Adapter 2.0 on each workstation, and SMC TigerStack 100 5324TX (non-switching Hub). The operating system used for measurement is SSS-CORE Ver. 1.1a [6].

The performance of MPICH 1.1 on SunOS 4.1.4 was also evaluated with the same equipments to make a comparison. MPICH employs TCP sockets for communication when it is used on a cluster of workstations.

In order to examine the effects of the write protocol, two different versions of MPI/MBCF are used for the performance evaluation. One issues requests for sending as explained in Sect. 2.1, and the other does not. The former implementation is denoted by SR and the latter NSR for short in the following. In NSR, the sender always transmits a message with *Memory-Based FIFO* and never with *Remote Write*, like the eager protocol.

From the performance parameters presented in Table 1, it is shown that the latency of MPI/MBCF is ten times smaller than that of MPICH/TCP. MPI/MBCF gains high bandwidth for small messages as well as for large messages. The latency of SR is smaller than that of NSR. In SR, however, request packets from the receiver interfere with message packets from the sender. Thus the bandwidth of SR is smaller than that of NSR.

Table 1. Basic performance parameters of MPI libraries with 100BASE-TX

MPI library	MPI/MBCF		MPICH/TCP
	SR	NSR	
round-trip time for 0 byte message (μ s)	71	112	968
peak bandwidth for 256 bytes message (Mbytes/s)	4.72	4.92	1.27
peak bandwidth for 16 Kbytes message (Mbytes/s)	10.15	10.34	5.59

3 Performance Evaluation with the NAS Parallel Benchmarks

3.1 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB) [1,2] is a suite of benchmarks for parallel machines. It consists of the following five kernel programs and three computational fluid dynamics (CFD) applications.

Table 2. Characteristics of NPB Programs with 16 processes

program	EP	MG	CG	IS	LU	SP	BT
communication freq (Mbytes/s)	0.00	20.21	36.24	22.10	5.58	19.50	14.68
communication freq (# of messages/s)	16	13653	6103	2794	5672	1657	2020
<i>Remote Write</i> availability rate (%)	50.51	0.02	47.89	97.15	8.66	42.55	45.22
performance ratio, MPI/MBCF vs. MPICH	1.01	1.42	1.30	1.46	1.36	1.59	1.19
performance ratio, SR vs. NSR	1.00	1.00	1.02	1.44	1.01	1.00	1.01

EP. Random number generation by the multiplication congruence method.

MG. Simplified multigrid kernel for solving a 3D Poisson PDE.

CG. Conjugate gradient method for finding the smallest eigenvalue of a large-scale sparse symmetric positive definite matrix.

FT. Fast-Fourier transformation for solving a 3D PDE.

IS. Large-scale integer sort.

LU. CFD application using the symmetric SOR iteration.

SP. CFD application using the scalar ADI iteration.

BT. CFD application using the 5×5 block size ADI iteration.

The NPB 2.x provides source codes written with MPI. Each of the eight problems is classified into five (S, W, A, B, and C) classes according to its problem size configuration.

3.2 Conditions

The same workstations as stated in Sect. 2.2 and 3Com Super Stack II Switch 3900 (switching Hub) were used. SR and NSR of MPI/MBCF on SSS-CORE Ver. 1.1a and MPICH 1.1.1 on SunOS 4.1.4 are compared.

We used gcc-2.7.2.3 and g77-0.5.21 for compilation. Since the FT kernel program cannot be compiled with g77, it is omitted in the following evaluation.

The problem size is fixed to Class W because the programs of Class A cannot be executed on SunOS for comparison owing to the shortage of memory.

3.3 Experimental Results

Table 2 shows the characteristics of the benchmark programs. These were measured on SR with 16 processes by inserting counter codes into MPI/MBCF. The communication frequency is computed from the total amount of data (or the total number of messages) transmitted among all processes divided by the execution time. The *Remote Write* availability rate is computed from the amount of data transmitted with *Remote Write*, divided by the total amount of data. The performance ratio is the reciprocal of the execution time ratio.

Table 3. Execution time of NPB EP in seconds

# of processes		1	2	4	8	16
SR	[speed-up]	120.94 [1.00]	60.50 [2.00]	30.26 [4.00]	15.14 [7.99]	7.57 [15.98]
NSR	[speed-up]	120.95 [1.00]	60.51 [2.00]	30.26 [4.00]	15.14 [7.99]	7.57 [15.98]
MPICH	[speed-up]	121.17 [1.00]	60.50 [2.00]	30.47 [3.98]	15.22 [7.96]	7.62 [15.90]

Table 4. Execution time of NPB MG in seconds

# of processes		1	2	4	8	16
SR	[speed-up]	39.30 [1.00]	23.08 [1.70]	13.52 [2.91]	7.31 [5.38]	4.80 [8.19]
NSR	[speed-up]	39.26 [1.00]	23.18 [1.69]	13.48 [2.91]	7.29 [5.39]	4.80 [8.18]
MPICH	[speed-up]	39.17 [1.00]	23.61 [1.66]	16.41 [2.39]	9.47 [4.14]	6.81 [5.75]

Table 3 shows the execution time of EP. 2^{26} random numbers are computed. In EP, interprocess communication occurs only for gathering final results. The amount of transmitted data is very small. Consequently the results of MPI/MBCF and MPICH differ by 1 % or less.

Table 4 shows the execution time of MG. The problem size is $64 \times 64 \times 64$ and the number of iterations is 40. Point-to-point communication operations for messages of around 1 Kbytes are performed very frequently to exchange data across partitioning boundaries. Since MPI/MBCF is suitable for fine-grain communication as shown in Sect. 2.2, the performance with MPI/MBCF is better by 42 % than that with MPICH. All of the receive functions in MG specify an unnecessary wild card `MPI_ANY_SOURCE` as a source. In SR, this disables the receiver from issuing requests for sending to a unique source. This causes extremely low availability (0.02 %) of *Remote Write*. Thus the results of SR and NSR differ very little.

Table 5 shows the execution time of CG. The problem size is 7000 and the number of iterations is 15. Collective reduction operations and point-to-point communication operations are performed for messages of around 10 Kbytes. Although the communication frequency of CG is high (i.e. hard for MPICH), the message size is large (i.e. easy even for MPICH). Therefore the difference between MPI/MBCF and MPICH in CG is smaller than in MG. *Remote Write* is applied to 48 % of messages in SR so that the performance of SR is improved by 1.7 % from NSR.

Table 6 shows the execution time of IS. The problem size is 2^{20} and the number of iterations is 10. About 1 Mbytes messages are exchanged at each iteration by collective all-to-all communication functions. Because the amount of computation is small, the performance of collective operations dominates the whole performance more and more as the number of processes increases. In MPI/MBCF, functions for collective operations are written so that receive functions are first

invoked. Thus, in SR, 97 % of messages are transmitted with *Remote Write*. The performance of SR is improved by 44 % from NSR and by 46 % from MPICH.

Table 7 shows the execution time of LU. The problem size is $33 \times 33 \times 33$ and the number of iterations is 300. Point-to-point communication operations are performed for messages of some hundred bytes. Since the message size is small, MPI/MBCF achieves better performance by 36 % than MPICH. As well as in MG, the use of `MPI_ANY_SOURCE` reduces the use of *Remote Write* to 8.7 % for SR.

Table 8 shows the execution time of SP. The problem size is $36 \times 36 \times 36$ and the number of iterations is 400. Point-to-point communication operations are performed for messages of around 10Kbytes. MPI/MBCF achieves better performance by 59 % than MPICH. There is little difference between results of SR and NSR, though SR utilizes *Remote Write* for 43 % of messages. This is because communication in SP is organized to hide latency in some degree.

Table 9 shows the execution time of BT. The problem size is $24 \times 24 \times 24$ and the number of iterations is 200. Point-to-point communication operations are performed for messages of around 10Kbytes. MPI/MBCF achieves better performance by 19 % than MPICH. As well as in SP, there is little difference between results of SR and NSR though SR utilizes *Remote Write* for 45 % of messages.

3.4 Summary of Results

Except for in EP, MPI/MBCF achieves better performance by 19 %–59 % than MPICH/TCP. Especially when small messages are transmitted frequently, the large overheads of MPICH/TCP lower the performance. Thus the difference in performance between two libraries expands, as shown in MG and LU.

The NSR implementation of MPI/MBCF uses *Memory-Based FIFO* alone. It has a resemblance to MPICH/TCP in that both of them are message-based implementations of MPI. The lower latency and higher bandwidth of NSR, which are mainly brought by MBCF, cause better performance by 2 %–60 % than MPICH/TCP on the very same machines.

Compared with NSR, SR utilizes *Remote Write* to communicate without buffering when receive functions are invoked before send functions (e.g. in CG, IS, and LU). Therefore SR achieves better performance in such cases. This shows that the combination of the eager protocol and the write protocol improves the practical performance, as well as the basic performance. When a wild card is specified as a source in the receiver, however, SR cannot use *Remote Write*. This deprives the opportunities of efficient communication in SR.

4 Summary

We have implemented a full MPI-1.2 library, MPI/MBCF, with a shared memory communication mechanism, MBCF. The performance of MPI/MBCF was evaluated on a cluster of workstations connected with a 100BASE-TX Ethernet.

Table 5. Execution time of NPB CG in seconds

# of processes		1	2	4	8	16
SR	[speed-up]	69.59 [1.00]	36.20 [1.92]	21.09 [3.30]	11.03 [6.31]	7.74 [8.99]
NSR	[speed-up]	69.59 [1.00]	36.84 [1.89]	21.44 [3.25]	11.38 [6.12]	7.87 [8.84]
MPICH	[speed-up]	68.05 [1.00]	38.70 [1.76]	23.90 [2.85]	12.74 [5.34]	10.06 [6.76]

Table 6. Execution time of NPB IS in seconds

# of processes		1	2	4	8	16
SR	[speed-up]	10.12 [1.00]	7.17 [1.41]	4.49 [2.25]	3.35 [3.02]	1.97 [5.14]
NSR	[speed-up]	10.08 [1.00]	7.15 [1.41]	4.82 [2.09]	3.77 [2.67]	2.83 [3.56]
MPICH	[speed-up]	8.89 [1.00]	7.10 [1.25]	5.41 [1.64]	4.73 [1.88]	2.88 [3.09]

Table 7. Execution time of NPB LU in seconds

# of processes		1	2	4	8	16
SR	[speed-up]	1038.72 [1.00]	535.36 [1.94]	277.25 [3.75]	149.58 [6.94]	80.89 [12.84]
NSR	[speed-up]	1028.93 [1.00]	538.76 [1.91]	281.93 [3.65]	151.02 [6.81]	82.08 [12.54]
MPICH	[speed-up]	1020.13 [1.00]	558.06 [1.83]	297.38 [3.43]	173.66 [5.87]	110.07 [9.27]

Table 8. Execution time of NPB SP in seconds

# of processes		1	4	9	16
SR	[speed-up]	1400.17 [1.00]	343.50 [4.08]	155.65 [9.00]	91.89 [15.24]
NSR	[speed-up]	1398.89 [1.00]	344.11 [4.07]	158.39 [8.83]	91.74 [15.25]
MPICH	[speed-up]	1392.28 [1.00]	407.87 [3.41]	201.55 [6.91]	146.46 [9.51]

Table 9. Execution time of NPB BT in seconds

# of processes		1	4	9	16
SR	[speed-up]	618.00 [1.00]	155.54 [3.97]	71.43 [8.65]	37.66 [16.41]
NSR	[speed-up]	618.71 [1.00]	155.82 [3.97]	67.78 [9.13]	37.87 [16.34]
MPICH	[speed-up]	616.76 [1.00]	197.20 [3.13]	91.39 [6.75]	44.89 [13.74]

By executing the NAS Parallel Benchmarks, it was shown that MPI/MBCF with *Remote Write* achieves better performance than MPI/MBCF without *Remote Write* and than MPICH/TCP. These results give corroborative evidence to our claim; a message passing library achieves higher performance by using a shared memory communication mechanism than with a message passing communication mechanism.

The experiments were made for two different combinations of operating systems and communication mechanisms on the very same machines with off-the-shelf hardware. These results show that it is possible to achieve large improvement of performance by improving the operating system and the communication library, without modifying applications. This also suggests the effectiveness of a cluster of workstations without dedicated communication hardware.

References

1. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March 1994.
2. D. Bailey, T. Harris, W. Saphir, R. Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, December 1995.
3. Message Passing Interface Forum. MPI: A message-passing interface standard. <http://www.mpi-forum.org/>, June 1995.
4. Message Passing Interface Forum. MPI-2: Extensions to the message-passing interface. <http://www.mpi-forum.org/>, July 1997.
5. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message-passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
6. T. Matsumoto, S. Furuso, and K. Hiraki. Resource management methods of the general-purpose massively-parallel operating system: SSS-CORE (in Japanese). In *Proc. of 11th Conf. of JSSST*, pages 13–16, October 1994.
7. T. Matsumoto and K. Hiraki. Memory-based communication facilities of the general-purpose massively-parallel operating system: SSS-CORE (in Japanese). In *Proc. of 53rd Annual Convention of IPSJ (1)*, pages 37–38, September 1996.
8. T. Matsumoto and K. Hiraki. MBCF: A protected and virtualized high-speed user-level memory-based communication facility. In *Proc. of Int. Conf. on Supercomputing '98*, pages 259–266, July 1998.
9. K. Morimoto. Implementing message passing communication with a shared memory communication mechanism. Master's thesis, Graduate School of University of Tokyo, March 1999.
10. K. Morimoto, T. Matsumoto, and K. Hiraki. Implementing MPI with the memory-based communication facilities on the SSS-CORE operating system. In V. Alexandrov and J. Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 223–230. Springer-Verlag, September 1998.

Performance and Predictability of MPI and BSP Programs on the CRAY T3E

J.A.González, C. Rodríguez, J.L.Roda., D.G. Morales,
F. Sande, F. Almeida, C. León

Departamento de Estadística, I.O. y Computación,
Universidad de La Laguna
Tenerife—Spain
{jaglez, jlroda, casiano}@ull.es

Abstract. It has been argued that message passing systems based on pairwise, rather than barrier, synchronization suffer from having no simple analytic cost for model prediction. The BSP Without Barriers Model (BSPWB) has been proposed as an alternative to the Bulk Synchronous Parallel (BSP) model for the analysis, design and prediction of asynchronous MPI programs. This work compares the prediction accuracy of the BSP and BSPWB models and the performance of their respective software libraries: Oxford BSPlib and MPI. Three test cases, representing three general problem solving paradigms are considered. These cases cover a wide range of requirements in communication, synchronisation and computation. The results obtained on the CRAY-T3E show not only a better scalability of MPI but that the performance of MPI programs can be predicted with the same exactitude than Oxford BSPlib programs.

1 Introduction

The computational Bulk Synchronous Parallel (BSP) model [3] considers a parallel machine made of a set of p processor-memory pairs, a global communication network and a mechanism for synchronizing the processors. A BSP calculation consists of a sequence of supersteps. The cost of a BSP program can be calculated by summing the cost of each separated superstep executed by the program; in turn, for each superstep the cost can be decomposed into: (i) local computation plus exchange of data and (ii) barrier synchronization. The two basic BSP parameters that model a parallel machine are: the gap g_p , which reflects per-processor network bandwidth, and the minimum duration of a superstep L_p , which reflects the time needed by all processors to synchronize as well as the time to send an unitary packet through the network. Table 1 shows these values for the Oxford BSPlib in the CRAY T3E.

Although BSP programs can be expressed using existing communication libraries as MPI, the counterpart is not always true. It has been stated that message passing systems based on pairwise, rather than barrier synchronization suffer from having no simple analytic cost for model prediction ([2] page 22). The **BSP Without Barriers**

model (BSPWB) introduced in [1] is a proposal to fill this gap. The execution of a MPI program in any processor consists of phases of computation followed by the necessary communications to provide and obtain the data for the next phase. Communication in this context means a *continuous stream* of messages. BSPWB divides a MPI program in what is called Message steps or "M-steps". In any M-step each processor performs some local computation, sends the data needed by the other processors and receives the data it needs for the next M-step. Processors may be in different M-steps at a given time, since no global barrier synchronization is used. We assume that:

- The total number of M-steps R , performed by all the p processors is the same.
- Communications always occur among processors in the same M-step s .

To achieve these two goals, the designer can arbitrarily divide the computation stages in "dummy M-steps" at any processor.

An approach to the actual MPI time $t_{s,i}$ when processor i finishes its s -th M-step is the value $\Phi_{s,i}$ given by the BSPWB model. We define the set $\Omega_{s,i}$ for a given processor i and M-step s as the set

$$\Omega_{s,i} = \{j / \text{Processor } j \text{ sends a message to processor } i \text{ in M-step } s\} \cup \{i\} \quad (1)$$

Processors in the set $\Omega_{s,i}$ are called "**the incoming partners of processor i in step s** ". The BSPWB time $\Phi_{s,i}$ of a MPI program is recursively defined by the formulas:

$$\Phi_{1,i} = \max \{w_{1,j} / j \in \Omega_{1,i}\} + (g h_{1,i} + L), \quad i = 0, 1, \dots, p-1 \quad (2)$$

$$\Phi_{s,i} = \max \{\Phi_{s-1,j} + w_{s,j} / j \in \Omega_{s,i}\} + (g h_{s,i} + L), \quad s=2, \dots, R, \quad i=0, 1, \dots, p-1$$

where $w_{s,i}$ denote the time spent in computing and $h_{s,i}$ denotes the number of packets communicated by processor i in step s :

$$h_{s,i} = \max \{in_{s,j} + out_{s,j} / j \in \Omega_{s,i}\}, \quad s = 1, \dots, R, \quad i = 0, 1, \dots, p-1 \quad (3)$$

and $in_{s,i}$ and $out_{s,i}$ respectively denote the number of packets incoming/outgoing to/from processor i in the M-step s .

Formula (2) says that processor i in its step s does not ends the reception of its messages until it has finished its computing time $\Phi_{s-1,i} + w_{s,i}$ and so have done the other processors j sending messages to processor i .

We define the **BSPWB packet size** as the size in which the time/size curve of the architecture has the first change in its slope. Special gap g_0 and latency L_0 values have to be used for messages of sizes smaller than the BSPWB packet size. The total time of a MPI program in the BSPWB model is given by

$$\Psi = \max \{ \Phi_{R,j} / j \in \{0, \dots, p-1\} \} \quad (4)$$

where R is the total number of M-steps.

Table 1. Latency and gap values of the BSP Model.

processors	2	4	8	16
L (sec)	5.46E-06	7.24E-06	1.03E-05	1.52E-05
g (sec/word)	5.30E-08	3.59E-08	3.33E-08	3.37E-08

The values of g , L_0 and L for the native versions of MPI in Table 2 were obtained using least square fit on the average time values of the communication times of different communication patterns and processor numbers.

Table 2. BSPWB Model parameters for the CRAY T3E.

	L_0 (sec)	L (sec)	g (sec/word)
BSPWB	1.00E-5	-2.11E-5	2.92E-8

2 Benchmarks: Prediction and Performance

2.1 Reduction Parallelism Test: Computing the Minimum Spanning Tree

Let $G = (V, E)$ be a graph with a set of N vertices V and a set of edges E . A graph is called a tree if it is a connected acyclic graph. A *spanning tree* of an undirected graph G is a subgraph $T = (V, E')$ of G that is a tree containing all the vertices of G . In a weighted graph each edge (i, j) is labeled with a weight $w[i][j]$. The weight of a subgraph is the sum of the weights $w[i][j]$ of the edges (i, j) in the subgraph. A minimum spanning tree *MST* for a weighted undirected graph is a spanning tree with minimum weight. The parallel version of the Prim algorithm to compute the *MST* presented in Figure 1 is a canonic example of the greedy technique.

```

1. void Prim(VertexSetType V_name , T, VertexType r)
2. {
3.   T = {r};
4.   d[r] = 0;
5.   for v in (V_name - T)
6.     d[v] = w[r][v];
7.   for i=2 to |V| {
8.     Let be u_name such that d[u_name] = min{d[v] / v in V_name - T};
9.     REDUCE(CompareDist, z, u_name, sizeof(VertexType));
10.    T := T ∪ {z};
11.    for v in (V_name - T)
12.      d[v] = min{d[v], w[z][v]};
13.  };
14.} /* Prim */

```

Fig. 1. Parallel pseudo-code for the MST.

The set of vertices V is partitioned among the p processors in disjoint subsets V_{name} of size N/p . The matrix w is distributed in such a way that processor $name$ contains the columns $w[i][j]$ for i in V_{name} and j in V . At the end of the algorithm each processor has a replica of the minimum spanning tree T . At any time, the vector $d[v]$ indexed in the vertices v of $V - T$ contains the minimum distance from v to the current tree T . The tree and the d vector are initialized in time $A + B * N/p$ in lines 3-6 for some constants A and B .

Instead of using as in standard BSP a single universal computational constant s , our analysis (both for BSP and BSPWB) of the computing times implies experimental evaluation. Maximal portions of code taking analytical constant time must be timed. The computational constants are obtained as average of those times.

On each stage i of the loop in lines 7- 13 a new vertex is introduced in the minimum spanning tree T . Each processor $name$ finds the nearest vertex u_{name} to the tree T (line 8) in time $C+D*N/p$. An all to all generic reduction on the commutative operation $CompareDist(u_i, u_j)$ is used to obtain the global nearest vertex z to the tree T . Both libraries MPI and Oxford BSPlib provide generic reductions routines: *MPI_Allreduce* in MPI and *bsp_fold* in the *Oxford BSPlib*. Thus, the differences in performance between the two implementations of the algorithm are basically due to the differences in efficiency of these two routines. Since the h -relations involved in the reduction correspond to one word, the time spent is dominated by the latency L . Although the insides of these routines are hidden to the programmer, we can expect the time of any smart implementation to behave logarithmically in the number of processors $R*log(P)+S = (R'+L+g)*log(P)+S$, where L and g are the Latency and gap of the respective models (L_0 for BSPWB). Constants R' and S reflect the computing overhead of the reduction subroutine. That overhead depends on the operation, and can not be neglected. Since the task of estimating the communication time inside the reduction can not be traced, we have instead estimated R and S by linear square fit in the logarithm of the processor number.

Once the nearest vertex z has been introduced in the tree (line 10), the distances $d[v]$ of the vertices v to the tree are computed in time $H*N/P+J$ in lines 11 and 12. Therefore, the total cost, both in the BSP and BSPWB models, is given by

$$A+B*N/p + C+D*N/p+ (R'+L+g)*log(P)+S+H*N/P+J \quad (5)$$

Table 3 corresponds to the average times of five executions of the former algorithm for a 4096 vertices graph on the CRAY-T3E at CIEMAT.

Table 3. Predicted and real times (Prim algorithm).

Processors	2	4	8	16
Predicted BSP	4.87	3.34	2.72	2.54
Real BSPlib	5.45	4.00	3.37	2.94
Predicted BSPWB	4.55	3.07	2.22	1.84
Real MPI	5.41	3.78	2.96	2.14

The generic reduction MPI routine is always faster than the corresponding Oxford BSP routine. Although it does not appear in the tables, the differences percentage $100*(MPIAlltoall-BSPfold)/MPIAlltoall$ between the time of the two reduce subroutines increases with the number of processors, going from 35% for two processors to 107% for sixteen processors.

The values of the computational constants appearing in this and the other two examples can be found in [ftp://ftp.csi.ull.es/pub/parallel/BSPWB/pvmmmpi99.html](http://ftp.csi.ull.es/pub/parallel/BSPWB/pvmmmpi99.html).

2.2 Data Parallelism Test. The Fast Fourier Transform

The pseudocode in Figures 2 and 3 show a divide and conquer (D&C) parallel algorithm computing the Discrete Fourier Transform. The code is essentially the same both for MPI and BSP. The algorithm has as input a replicated vector of complex A , and the number n of elements. The result vector B is also required to be in all the processors. The variable *NUMPROCESSORS* holds the number p of available processors. If there is only one processor, a call to the sequential algorithm *seqFFT* solves the problem in time $C*n/p*log(n/p)+D$. Otherwise, the algorithm tests if the problem is trivial. If not, function *odd_and_even()* decompose the input signal A in its odd and even components in time proportional to the size of the vector. After that, the *PAR* construct in line 15 makes two parallel recursive calls to the function *parFFT*. The results of these computations are returned in $A1$ and $A2$ respectively. The time spent in the successive divisions is:

$$(A*n/2+B)+(A*n/4+B)+...+(A*n/p+B) = A*n*(p-1)/p + B*log(p) \quad (6)$$

```

1. void parFFT(Complex *A, Complex *B, int n) {
2.   Complex *a2, *A2;          /* Even terms */
3.   Complex *a1, *A1;          /* Odd terms */
4.   int m, size;
5.
6.   if(NUMPROCESSORS > 1) {
7.     if (n == 1) {             /* Trivial problem */
8.       b[0].re = a[0].re;
9.       b[0].im = a[0].im;
10.    }
11.    else {
12.      m = n / 2;
13.      size = m * sizeof(Complex);
14.      Odd_and_even(A, a2, a1, m);
15.      PAR(parFFT(a2,A2,m), A2,size,
16.          parFFT(a1,A1, m), A1,size);
17.      Combine(B, A2, A1, m);
18.    }
19.  }
20.  else
21.    seqFFT(A, B, n);
22. }

```

Fig. 2. Code for the FFT.

The macro in Figure 3 divides the set of processors in two groups, G and G' . Each processor *name* in a group G chooses a *partner* in the other group G' , and the results are interchanged between partners. The first exchange of the two vectors of size n/p marks the end of the first superstep in BSP and the first M-step in BSPWB (lines 7 and 11 of figure 3). Consequently, the time of this first step is given by:

$$A*n*(p-1)/p + B*log(p) + (C*n*log(n/p)/p+D) + h_M*g_M+L_M \quad (7)$$

Where g_M and L_M denote the gap and latency in the parallel computing model M and h_M is the h -relation size: $h_{BSP} = n/p$ for BSP and $h_{BSPWB} = 2*n/p$ for BSPWB.

The $\log(p)$ successive calls to the function *Combine()* (line 17) compute the Fourier transform of the input vector in time $E*m+F$ proportional to the sizes m of the auxiliary vectors $A1$ and $A2$. Unless for the last combination, the first $\log(p)-1$ combinations are followed by the swapping of the results. Thus, we have $\log(p)$ steps with cost:

$$(E*(p-1)*n/p+F*\log(p))+g_M*\&_M*n*(p-2)/p+(\log(p)-1)*L_M \quad (8)$$

where the factor $\&_M = 1$ for BSP and $\&_M = 2$ for the BSPWB.

```

1. #define PAR(f1, r1, s1, f2, r2, s2) {
2.   int partner;
3.   NUMPROCESSORS /= 2; BIT--; /*BIT==log(NUMPROCESSORS)*/
4.   partner = name ^ (1 << BIT);
5.   if ((name & (1 << BIT)) == 0) {
6.     f1;
7.     SWAP(r1,s1,partner,r2,s2);
8.   }
9.   else {
10.    f2;
11.    SWAP(r2,s2,partner,r1,s1);
12.   }
13.   NUMPROCESSORS *= 2; BIT++;
14. }
```

Fig. 3. Code of the macro PAR.

Table 4. Predicted and real times (FFT algorithm).

Processors	2	4	8	16
Predicted BSP	5.29	2.97	1.88	1.36
Real BSPlib	5.11	2.95	1.92	1.45
Predicted BSPWB	5.29	2.98	1.89	1.38
Real MPI	5.29	2.99	1.90	1.38

An overlapping of the predicted and actual times on both software platforms is observed. As for the former experiment, the better performance of the MPI version grows with the processor number. The percentage error of the times predicted by the BSPWB model for MPI is less than 1%, which is smaller than the obtained using the BSP model for the Oxford BSPlib. This is a remarkable result when considering that the values of g and L used in BSPWB do not depend on the number of processors.

2.3 Pipeline Parallelism Test. The 0-1 Knapsack Problem

The 0-1 Knapsack Problem (KNP) is defined by a set of N objects of given weights w_i an profits p_i and a knapsack with a limited capacity M . The goal is to find a subset of objects fitting in the knapsack and maximizing the total profit. Let us denote by $Q[k][m]$ the optimal income for the KNP subproblem constituted by the first k objects and a knapsack of capacity m . Applying the Dynamic Programming Principle to this problem leads to the following state equation:

$$Q[k][m] = \max \{Q[k-1][m], Q[k-1][m-w_k] + p_k\}, N \geq k \geq 1, m \geq w_k \quad (9)$$

$$Q[k][m] = Q[k-1][m], N \geq k \geq 1, m < w_k$$

$$Q[0][m] = 0, m = 0, 1, \dots, M$$

A parallel pipeline algorithm using N processors can be easily designed taking advantage of the fact that dependencies in formula (9) occur among adjacent values of k . Each processor *name* takes charge of computing the values of a different row $Q[name]$. Figure 4 presents the parallel pseudocode.

```

1.  int KNP(int M, int Last) {
2.    int *Q, q, c, i;
3.    Q=(int*)calloc(M+1, sizeof(int));
4.    for(c=0;c<=M;c++) {
5.      receive(PREVIOUS,&q);
6.      Q[c]=max(Q[c], q);
7.      i = c+Wname;
8.      if (i ≤ M) Q[i]=max(Q[i], q+pname);
9.      if (!Last) send(NEXT, &Q[c]);
10.     /* barrier if BSP */
11.   }
12.   q=Q[M];
13.   free(Q);
14.   return q;
15. }
```

Fig. 4. Pipeline algorithm for the KNP.

Through the call to the *calloc* function in line 3, each processor *name* initializes to zero its vector of optimal values Q in time $A_1 * M + B_1$. The computation inside the loop from lines 5 to 10 takes constant time A_2 . To optimize communications, the MPI implementation combines the send at line 9 and the receive at line 5 in a single *MPI_Sendrecv* operation. Thus, each iteration of the loop gives place to a new step. The time taken by the *h*-relation is dominated by the synchronization/latency time. Neglecting the contribution of the message size, the loop time can be approached by $(M+1)*(A_2+L)$. Denoting by A_3 the constant time consumed in lines 12-14, we can write the time of the code in figure 4 as:

$$A_1 * M + B_1 + (M+1)*(A_2+L) + A_3 \quad (10)$$

The N virtual processors of the former algorithm have been mapped into the p physical processors using a cyclic policy. The computation is divided in N/p stages. On stage b processor k virtualizes processor *name* = $b * N/p + k$. In the MPI version the BSPWB time complexity is increased by the virtualization factor N/p plus the time $L + (p-2)*(A_2+L)$ necessary to load the pipe:

$$L + (p-2)*(A_2+L) + \{A_1 * M + B_1 + (M+1)*(A_2+L) + A_3\} * N/p \quad (11)$$

Since the pipe is loaded at the beginning of each band, the BSP time of the algorithm is given by:

$$\{A_1 * M + B_1 + (M + I + (p - 1)) * (A_2 + L_p) + A_3\} * N / p \quad (12)$$

Table 5. Actual and Predicted times for BSP and BSPWB for the KNP.

Processors	2	4	8	16
Predicted BSP	27.73	15.64	9.34	5.92
Real BSPlib	30.95	16.83	9.46	6.62
Predicted BSPWB	36.75	18.37	9.19	4.59
Real MPI	34.61	17.67	9.03	4.86

As it was observed in the FFT case, the Oxford BSPlib version is faster up to 4 processors. However, for 8 and 16 processors the MPI version presents a better time. Lets again emphasize that although the values of g_p and L_p in the BSP prediction formulas have been parameterized in the number p of processors, the errors observed in BSP are larger than the observed for the BSPWB model.

3 Conclusions

The results obtained for the three considered parallel applications not only show a better scalability of MPI but that the performance of MPI programs can be predicted with the same exactitude than the performance of Oxford BSPlib programs. However, for the last two cases, the Oxford BSPlib is slightly faster when the number of processors is small and for the Prim example the times are similar. The performance of MPI is better for the three cases when the number of processors increases. The differences both in the performance and in the precision can not be considered substantial.

Acknowledgements

We wish to thank to CIEMAT and to CCCC for allowing us the access to their resources.

References

- 1 Roda J., Rodríguez C., Morales D.G., Almeida F., Pulido P., Dorta D. *Breaking the Barriers: Two Models for MPI Programming*. Proc. of Int. Conference on Parallel Architectures and Compilation Techniques. pp. 248-255, 1998.
- 2 Skillcorn, D.B., Hill, J., McColl, W.F. *Questions and Answers about BSP*. Oxford University Computing Laboratory. Report PRG-TR-15-96. 1996.
- 3 Valiant L.G. *A Bridging Model for Parallel Computation*. Communications of the ACM, 33(8). pp. 103-111, 1990.

Automatic Profiling of MPI Applications with Hardware Performance Counters

Rolf Rabenseifner*

Center for High Performance Computing (ZHR), Dresden University of Technology
Zellescher Weg 12, Willers-Bau A 117, D-01062 Dresden, Germany
www.tu-dresden.de/zhr/, rabenseifner@rus.uni-stuttgart.de

Abstract. This paper presents an automatic counter instrumentation and profiling module added to the MPI library on Cray T3E and SGI Origin2000 systems. A detailed summary of the hardware performance counters and the MPI calls of any MPI production program is gathered during execution and written in `MPI_Finalize` on a special `syslog` file. The user can get the same information in a different file. Statistical summaries are computed weekly and monthly. The paper describes experiences with this library on the Cray T3E systems at HLRS Stuttgart and TU Dresden. It focuses on the problems integrating the hardware performance counters into MPI counter profiling and presents first results with these counters. Also, a second software design is described that allows the integration of the profiling layer into a dynamic shared object MPI library without consuming the user's PMPI profiling interface.

Keywords: MPI, Counter Profiling, Instrumentation, Hardware Performance Counters, Trace-based Profiling, PerfAPI, PCL.

1 Counter-Based Profiling

Today, job accounting on MPP hardware platforms does not provide enough information about the computational efficiency or about the efficiency of message passing (MPI) usage either to the users or to the computing centers. There is no information available about bandwidth and latency or integer and floating point operation rates achieved in real application runs. Therefore, users and hotline centers have no reliable information base for technical and political decisions with respect to programming and optimization investment. Existing trace-based profiling tools are too complicated for a first glance at an application and can be used in small test-jobs only, not in long-running production jobs.

To solve this problem, the High-Performance Computing-Center (HLRS) at the University of Stuttgart has combined the method of counter-based profiling with the technics of writing system log-files. For each MPI routine, the number

* The author is an employee of the High-Performance Computing-Center (HLRS) at the University of Stuttgart (www.hlrs.de/people/rabenseifner/). Most of this work was done while the author was a visiting research associate at the ZHR from January to April 1999.

of calls, the time spent in the routine and the number of transferred bytes are written at the end of each parallel job to a syslog file of the computing center and, optionally, to a user file. The integration of the PCL library [1] allows the automatic instrumentation with the microprocessor's hardware performance counters (e.g. floating point instructions) to get information about the computational efficiency of each program. With that, the user has a criterion whether tuning the numerical part or the communication part promises greater benefit.

An analysis tool reads the syslog file and, on a weekly basis, sends a summary to each user about her/his jobs and writes a web-based summary for the computing center. The results of the first half-year on CRAY T3E 900-512 at the HLRS are presented in [8]. In a survey, our users showed that in the past, the profiling information was used only seldom for tuning the individual applications because the profiling tool was only available after the application development was finished and production was started. But 75 % of those interviewed believe that the profiling can help in the future to improve their applications [9].

The profiling was implemented, tested and installed as default library on the T3E systems at HLRS Stuttgart and TU Dresden, and it is now ported to the Origin2000 at TU Dresden. The counter-based profiling only has a minimal overhead. The memory requirements on a T3E-900 are 200 kBytes. The counting requires 0.3 - 0.5 μ sec per MPI call and writing the syslog file requires about 0.1 sec for each job. The overhead was 0.03 % of the application CPU time on the first half-year average in Stuttgart. Including the hardware counters, the overhead is about 300 kBytes memory, 2 μ sec/call and about 0.1 - 0.2 % (expected) in all.

The PCL library was developed by the Forschungszentrum Jülich. For integrating the PCL library, the hardware counters' reading routine of the PCL library on the T3E has been optimized from about 35 μ s to about 0.5 - 1.0 μ s by removing the operating system calls. This allows differentiation between counting hardware events inside and outside of the MPI routines. This is important because otherwise some hardware counters (load, integer instruction, any instructions) could not be used for measuring the user application since the busy wait operations of MPI would inflate their values.

2 Software Design

To use the instrumented MPI library as default library, it is necessary to export the full MPI and PMPI interface for Fortran and C, as described in the MPI standard and implemented in the public and vendor's MPI libraries. This means that it was not possible to consume the PMPI profiling interface for the intended instrumentation, i.e. a method had to be used that allowed two profiling layers. The Figures 1 and 2 show the different software designs for Unix libraries (archive libmpi.a) and dynamic shared objects (DSO libmpi.so).

For Unix libraries, the instrumentation is implemented as one wrapper routine to each MPI routine and added to the original MPI library, in which the original routines' names are modified with a binary-file editor. This interface was developed for a CRAY T3E system and is described in [8].

For DSOs, the new MPI-DSO libmpi.so contains only the instrumented wrappers and an initialization routine that binds the (up to this time) unresolved

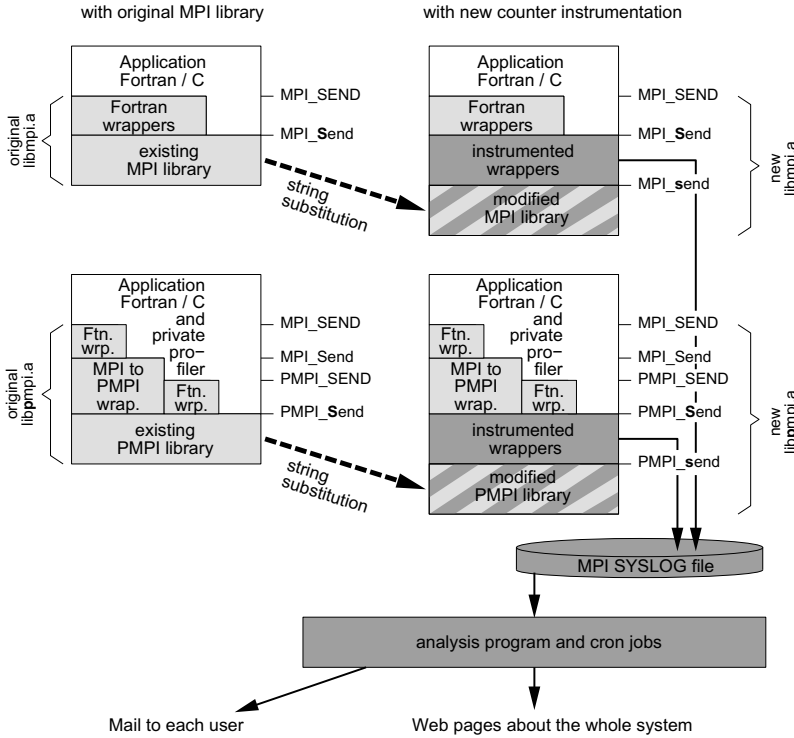


Fig. 1. The software design for MPI libraries (libmpi.a)

MPI references of the wrapper routines with `dlopen` and `dlsym` to the original MPI-DSO at start-up time. For this, the original MPI-DSO library was renamed `libMpi.so`. The dynamic linking at start-up time is necessary because static linking is impossible since the entry-point names of the instrumented wrapper routines and of the original MPI library routines are identical. This design requires that the original MPI routines never internally call other MPI routines. This is the case for the SGI MPI library. For applications written in C, this design adds only one additional subroutine call and the instrumentation.

For applications written in Fortran, the design depends on the implementation method of the Fortran MPI language binding. If a Fortran MPI routine is implemented as a Fortran-to-C wrapper routine that calls its C counterpart, then an empty wrapper for this Fortran interface must be added to the new `libmpi.so` and the dynamic linking establishes the following calling stack: application → new empty Fortran wrapper → original Fortran-to-C wrapper → new instrumented C wrapper → original MPI C interface. Compared with C, this case costs one additional subroutine call. If a Fortran MPI routine is implemented directly, then the Fortran wrapper in the new `libmpi.so` must be instrumented like the C wrapper, and there are no additional costs. This case may be necessary for routines with arguments that are externals or for optimized MPI routines. Additional wrappers must be included for the three special argu-

reading, and b) they can be *read out*, i.e. they are always reset to zero after reading them. For both interfaces, the instrumented wrappers have to implement one integer operation for each hardware counter before calling the original MPI routine and another one after returning from it:

In case a) `events_in_mpi - = counter` must be issued before each MPI call and `events_in_mpi + = counter` after its return and `events_in_application = -counter` at the beginning and `events_in_application + = (counter - events_in_mpi)` at the end of the whole application.

In case b) `events_in_application + = counter` must be issued before each MPI call and `events_in_mpi + = counter` after its return and `counter = events_in_mpi = events_in_application = 0` at the beginning and `events_in_application + = counter` at the end of the whole application. This means that the major requirement of a common **low-level** interface to access the micro-processors' hardware counters is that the two operations plus and minus must exist in the form

$$\begin{aligned} &\text{for (i=0; i<number_of_hardware_counters; i++)} \\ &\quad \text{local_event_counter[i] } \pm = \text{hardware_counter[i]} \end{aligned} \quad (1)$$

and also the information must be available whether this is a *read* or *read out*.

Unfortunately, neither the PCL library nor the PerfAPI interface design meet this requirement. E.g., the PCL library only has a *read out* interface, which adds additional operations, cache misses and resets of the hardware counter if the hardware supports the *read* interface. Additionally, PCL only has a high level interface that implements a matrix operation on the set of hardware counters and that implies at least an additional load/store for each counter. This matrix operation implements the mapping of the counters defined by PCL to any hardware counter or any difference or sum of several hardware counters. The matrix operation would be done twice for each MPI wrapper call, if PCL were used, instead of only once at the end of the application, which happens if the method (1) is used and the matrix operation is done only once before writing the results of `events_in_mpi` and `events_in_application` to the syslog-file. To integrate the hardware performance counters into the automatic MPI profiling, we have added an interface to the PCL library for the CRAY T3E that is similar to the required interface (1). With this and by removing all unnecessary operating system calls, the time to access the hardware performance counters could be reduced from 35 μs to about 0.5-1.0 μs .

4 First Results with Hardware Counters

Fig. 3 shows the users' profiles in the first seven weeks we used the hardware performance counters. Each row represents one user. The upper and lower part plot the same information, but differently sorted. The upper part is sorted by the CPU time consumed by each user, and the lower part by the total instruction rate. Each part combines three different plots:

The solid line in the left diagram represents the CPU time each user has consumed as part of the total time of the whole system in the analyzed time interval. The users are sorted by this value. The vertical bar marks 0.5 % of the

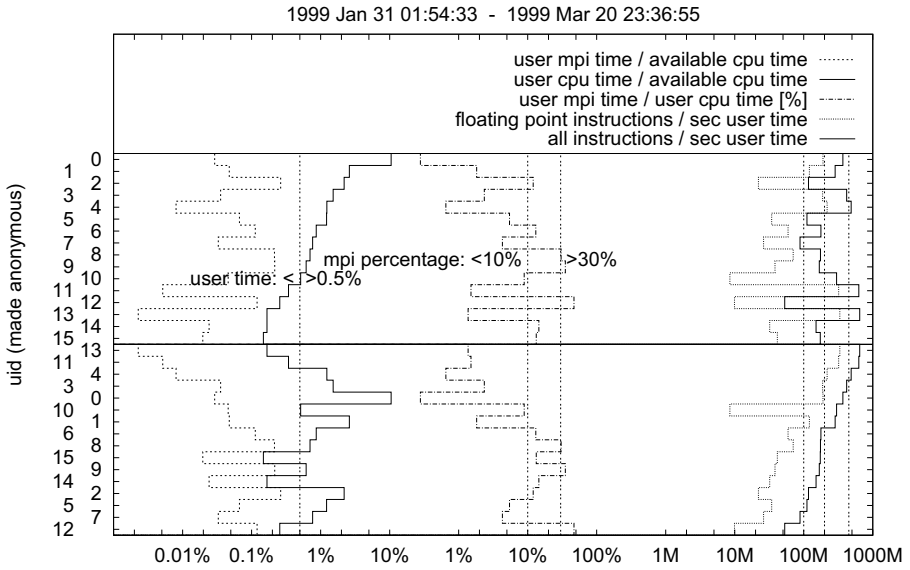


Fig. 3. Users' profiles, sorted by CPU time (upper part) and by instruction rate (lower part) – description see Section 4

total system. The figure represents the top 16 users of the HLRS that have used the new MPI library which was now instrumented with the hardware counters. In total, the 16 users have consumed 24.3 % of the whole system. The left dashed line shows the percentage of the time the user applications have consumed in MPI routines.

The diagram in the middle shows the ratio of MPI time to application time. The vertical bars mark a ratio of 10 and 30 %. The MPI percentage varies between 0.3 % and 46.8 %. On average, the MPI percentage was 5.2 %.

The diagram on the right shows the floating point instruction rate (dotted line) and the total instruction rate (solid line). The numbers are averages referring to one processor. On a T3E, each floating point instruction can execute one or two floating point operations. The floating point operation rate cannot be measured, but lies between the floating point instruction rate and twice the floating point rate. About half of the users' application runs could be used to analyse the hardware counters. The other jobs could not read the counters because the application was not yet relinked with the new library or else the partition was moved to other processors during execution. The floating point instruction rate of these 16 users varies between 9 and 333 MFLinstructions/sec (MFLips); weighted with the CPU time, the average is 138 MFLips, which implies that the MFLOP rate is between 138 and 276 MFLOPS, i.e. between 15 and 30 % of the peak performance of 900 MFLOPS. The vertical bars mark 100, 200 and 450 MFLips. The total instruction rate of the application code, except the MPI routines (solid line), is computed by dividing the number of instructions in the whole application minus the number of instructions executed in the MPI

routines by the whole execution time. The total instruction rate varies between 53 and 647 M_instructions/sec.

The upper part of the picture helps to review the efficiency of the most relevant users. The lower part of the picture gives an insight into the correlation of MPI percentage and instruction rate. The numbers presented for each user (i.e. the MPI percentage, the floating point instruction rate and the total instruction rate) are a major information base for decisions with respect to programming and optimization investment since these numbers give a good overview of the achieved efficiency in computation and communication. The analysis tool sends these numbers and additional details to each user in a weekly mail.

5 Related and Future Work

[1] describes the hardware counter library PCL. [5] is a standardization effort for accessing of the hardware performance counters. [10] is a comprehensive overview about monitoring and profiling systems. Trace-based profiling and analysis is described in [2,3,6,7]. [4] describes a local, user-callable MPI counter profiling. [8] focuses on the global view of the counter-based MPI profiling and includes the statistical results of half a year of profiling nearly all MPI applications running on a CRAY T3E 900-512, but without instrumenting the applications with the hardware counters. [9] describes the scalability of the profiling user interface.

We hope that the results of this project can influence the standardization effort of PerfAPI and that the new low-level interface for PCL can be implemented on all supported hardware platforms in an efficient way. Also, it is planned to use the hardware counter profiling for *all* applications and not only for MPI applications, although there is no plan to differentiate then between hardware events issued by the application code and those issued by non-MPI communication (e.g. with PVM, HPF or shmem). We will generate global half-year statistics of the CRAY T3E 900-512 used by the HLRS Stuttgart, similar to that in [8] but including the hardware counters. The major goal is to extend this profiling to all terminating correctly applications to see the hardware performance counters not only on MPI applications. The computing center has planned to use the profiling results to offer the users help in optimizing their individual applications. In a survey, we saw that 85% of our users would like to be addressed for that reason [9]. Extending the profiling interface for OpenMP applications is under investigation.

6 Conclusion

This project shows that combining the methods of counter profiling, job accounting and accessing the hardware performance counters can give more insight into the users' applications than achievable by previously used tools with similar costs. The paper has shown two different methods to integrate an additional profiling level into existing MPI archives and dynamic shared objects without losing the standardized MPI profiling interface for other profiling tools. The automatic MPI profiling is a method to get enough information to decide whether

the application is running as expected, one more chance to detect major bottlenecks and a basis to decide whether trace based tools should be used to optimize the communication pattern or whether direct hardware counter instrumentation should be applied to optimize the computational part.

Acknowledgments

The author would like to acknowledge his colleagues at ZHR and HLRS and all the people that supported this project with suggestions and helpful discussions. He would like to thank especially R. Berrendorf for his support of the PCL library, W.E. Nagel for productive discussions and for the invitation to Dresden, R. Rühle for giving major resources for this project, and M. Heine and E. Salo for the hints on SGI's MPI/DSO.

References

1. R. Berrendorf and H. Ziegler, *PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors*, internal report FZJ-ZAM-IB-9816, Forschungszentrum Jülich, Oct. 1998. <http://www.fz-juelich.de/zam/docs/autoren98/berrendorf3.html>
2. M. T. Heath, *Recent Developments and Case Studies in Performance Visualization using ParaGraph*, Proceedings of the Workshop Performance Measurement and Visualization of Parallel Systems, G. Haring and G. Kotsis (ed.), Moravany, Czechoslovakia, Oct. 1992, p. 175-200.
3. V. Herrarte and E. Lusk, *Studying Parallel Program Behavior with Upshot*, Argonne National Laboratory, technical report ANL-91/15, Aug. 1991
4. *HP MPI User's Guide*, 4.1 Using counter instrumentation, HP, B6011-90001, Third Ed., June 1998.
5. P. J. Mucci, S. Browne, G. Ho and C. Deane, *PerfAPI - Performance Data Standard and API*. <http://icl.cs.utk.edu/projects/papi/>
6. W. E. Nagel et al., *VAMPIR: Visualization and Analysis of MPI Resources*, Supercomputer 63, Volume XII, Number 1, Jan. 1996, pp. 69-80. <http://www.kfa-juelich.de/zam/docs/autoren95/nagel2.html> and Technical Report KFA-ZAM-IB-9528, Jan. 1996
<ftp://ftp.zam.kfa-juelich.de/pub/zamdoc/ib/ib-95/ib-9528.ps>
7. W. E. Nagel and A. Arnold, *Performance Visualization of Parallel Programs: The PARvis Environment*, technical report, Forschungszentrum Jülich, 1995. <http://www.fz-juelich.de/zam/PT/ReDec/SoftTools/PARtools/PARvis.html>
8. R. Rabenseifner, *Automatic MPI Counter Profiling of All Users: First Results on a CRAY T3E 900-512*, Proceedings of the Message Passing Interface Developer's and User's Conference 1999 (MPIDC'99), Atlanta, USA, March 1999. www.hlrs.de/people/rabenseifner/publ/publications.html
9. R. Rabenseifner, S. Seidl and W. E. Nagel, *Effective Performance Problem Detection of MPI Programs on MPP Systems: From the Global View to the Detail*, Parallel Computing '99 (ParCo99), Delft, the Netherlands, August 1999. www.hlrs.de/people/rabenseifner/publ/publications.html
10. M. van Riek, B. Tourancheau and X.-F. Vigouroux, *Monitoring of Distributed Memory Multicomputer Programs*, University of Tennessee, technical report CS-93-204, and Center for Research on Parallel Computation, Rice University, Houston Texas, technical report CRPC-TR93441, 1993. <http://www.netlib.org/tennessee/ut-cs-93-204.ps> and <ftp://softlib.rice.edu/pub/CRPC-TRs/reports/CRPC-TR93441.ps.gz>
11. The Parallel Tools Consortium – www.ptools.org

Monitor Overhead Measurement with SKaMPI

Dieter Kranzlmüller¹, Ralf Reussner², and Christian Schaubschläger¹

¹ GUP Linz, Joh. Kepler University Linz, Austria
[kranzlmüller|schaubschlaeger]@gup.uni-linz.ac.at

² LIIN, University of Karlsruhe, Germany
reussner@ira.uka.de

Abstract. The activities testing and tuning of the software lifecycle are concerned with analyzing program executions. Such analysis relies on state information that is generated by monitoring tools during program runs. Unfortunately the monitor overhead imposes intrusion onto the observed program. The resulting influences are manifested as different temporal behavior and possible reordering of nondeterministic events, which is called the probe effect. Consequently correct analysis data requires to keep the perturbation a minimum, which defines the need for monitors with small overhead. Measuring the actual overhead of monitors for MPI programs can be done with the benchmarking suite SKaMPI. It's results serve as a main characteristic for the quality of the applied tool, and additionally increase the user's awareness of the monitoring crisis. Besides that, the measurements of SKaMPI can be used in correction algorithms, that remove the monitoring overhead from perturbed traces.

1 Introduction

Software analysis is concerned with detecting errors and performance bottlenecks in programs. Since program execution is of main interest, most analysis activities rely on process state information that is generated by observation tools during representative program runs. Yet, this observation also introduces problems, which are summarized as the *probe effect* [2]. This means that monitoring a program influences its behavior, and the generated analysis data does not represent the same execution as if monitoring is turned off.

The monitor perturbation takes place in time and space [7], because it depends on the time spent in monitoring functionality and the amount of memory required for the monitoring code and the observation data. The memory consumption is not a problem as long as enough memory resources are available. More critical is the timing perturbation, which occurs in two areas: Primarily, the occurrence time of each observed event is delayed by the amount of time spent in the monitoring functions. Additionally, the different event times may sometimes lead to variations in event ordering if nondeterministic programs are considered.

The consequences are manifold. Firstly, the observed execution is different from the same program run without observation. Therefore it is difficult to connect both executions and obtain the required information. Secondly, the observation may even introduce a completely different execution path and therefore

different results. Additionally, errors and bottlenecks may be hidden in paths not taken by the monitored execution, while errors may be introduced, that would never occur in the original program.

It is therefore required to raise the user’s awareness about the probe effect and to integrate solutions in corresponding analysis tools. Yet, so far tool developers either neglected the importance of the monitor overhead or, if they included measurements, they did not provide information about how this measurements were performed. With SKaMPI [9], the Special Karlsruher MPI-Benchmark, we provide an approach to standardize these measurements, which allows to compare different monitoring tools. While SKaMPI was originally intended for benchmarking of MPI implementations, it can easily be extended for other measurements as described in this paper.

The paper is organized as follows. In the next section we define our target of investigation, which are nondeterministic MPI programs. For simplification we will focus only on point-to-point communication. Afterwards we define the amount of monitor overhead for a generally valid and abstract monitoring approach. Section 4 introduces SKaMPI and its extensions for monitor overhead measurements as well as some results for an example implementation.

2 Nondeterministic MPI Programs

The main focus of our research are parallel programs based on the standard Message Passing Interface MPI [8]. In MPI parallelism is achieved with multiple instances of (possibly the same) code that are executed concurrently. Synchronization and communication is performed explicitly via dedicated functions provided by the corresponding MPI library. For example, to exchange messages between two processes one could use the basic functions `MPI_Send` and `MPI_Recv`.

```
MPI_Send(buf, count, datatype, dest, tag, comm)
MPI_Recv(buf, count, datatype, source, tag, comm, status)
```

With these functions, the message at memory location `buf` is transferred between two processes, if the variables are set accordingly: At the sending process `dest` points to the receiving process, while at the receiving process `source` identifies the sender. Additionally the parameter `tag` must fit, which serves as an indicator for the contents of the message. Other parameters are provided for additional functionality.

As extension to the standard functionality the definition of `MPI_Recv` introduces nondeterminism, because it is possible to accept messages originating from more than one process at a particular receive. This is achieved with wild cards, where the identifier for the message source is specified as `MPI_ANY_SOURCE`, while the tag may be defined as `MPI_ANY_TAG`. In fact, if there is more than one message racing towards a receive, the order of arrival may be determined by system characteristics like processor speed, scheduling and caching strategy, or contention on the communication network. The behavior of the process after the receive may be affected and is therefore unpredictable, even if the same input is

provided for the program. Yet, due to performance improvements such behavior may be requested by the user, e.g. in case of implementing a FIFO-queue.

Besides its useful features, nondeterminism also introduces a major drawback for observation tools. Since monitors perturb the target programs, the occurrence of messages at wild card receive events may be influenced. It is possible, that the arrival order of messages in the original program is completely different from the arrival order in the monitored execution. Additionally, the results of the program are possibly changed, if the program code after the wild card receive depends on the arrival order. As a consequence, errors may be hidden in branches of the code, that are not taken if the program is observed. Furthermore, less critical errors may be emphasized, which may possibly never occur with the original program.

3 Monitor Overhead

The problems as described above occur, whenever an observer is added to a system, and are therefore not solely connected to message-passing programs. To react to the probe effect's implications, it is necessary to identify the amount of overhead that is generated by the observer. Only with known intrusion is it possible to reduce the generated overhead, to provide tools for overhead correction, and to increase the user's awareness of this problem. To identify the amount of overhead, it is important to determine the places where monitoring occurs. Of main interest is the group of events, that are observed during execution. An event is an action without duration that takes place at some time during the execution and changes the state of a process [6]. The reason for an event is a programming statement, that has been placed during the implementation. Clearly, not every programming statement might be interesting, and thus, the numbers of observed events is limited and defined either by the user or the applied analysis tool.

For the remainder of this paper we will focus on communication events in message-passing programs, especially those generated by the `MPI_Send` and `MPI_Recv` function as described in section 2. The important questions are, how monitoring code is added to these function calls during the instrumentation phase, and where monitoring functionality interferes with the application's execution.

Instrumenting a program can be done in various ways. One example is to manually or automatically substitute interesting function calls with corresponding calls from the monitoring library. More sophisticated techniques, like binary wrapping [1], may apply changes by patching object files. Another solution is the profiling interface defined by the MPI standard itself, which requires a simple name-shifting convention from any MPI implementation [8]. It allows to integrate monitoring functionality without the need to access the MPI source code.

After adding monitoring code to the target application, activities of the monitor are performed whenever an instrumented statement or function call is executed. In principle, the interaction between monitor and application is as described in Figure 1. Instead of calling the MPI library directly from the ap-

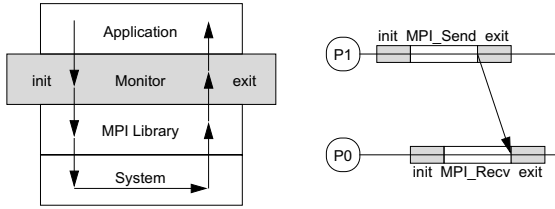


Fig. 1. Monitor overhead in communication functions

plication, the application calls the monitor function, which on the other hand initiates the original MPI function. After the MPI function is finished, it returns again to the observer which notices the end of the function and performs analysis of the return parameters and error codes. This defines the two areas of monitor activities, the init part and the exit part as displayed in Figure 1. Please note, while it is not necessary to perform monitoring functions before and after the original function call, it is quite common in existing monitors to detect successful function completion.

As soon as the application has been instrumented and an execution has been performed, observation data is available. Additionally, the produced monitor overhead can be evaluated. The amount of the overhead is mainly determined by the functionality that is performed in the init and the exit part, which can be characterized by [3]:

- the amount of data to be processed
- the number of measurements to be performed
- whether source code connection has to be traced or not
- the connection between the monitoring tool and the observer

The amount of data and the number of measurements is clearly defined by the requirements of the analysis task. For example, if users are only interested in a correct logical communication structure, no timing measurements need to be performed. Similarly, the source code connection is only needed in some cases.

Another main characteristic is the connection between the monitor and the analysis tool. While the monitor has to be active during program execution, the analysis task may be performed concurrently or at another time and possibly at another location. This defines the two possible connections: on-line and post-mortem. The advantage of on-line connection is that program behavior and monitor characteristics can be interactively determined by the user. Yet, this may also mean a bigger overhead, since user interaction may delay the program significantly. On the other hand, post-mortem mechanisms require a lot of storage space and permit only limited program execution steering. However, since it is not possible to estimate user activities, we focus mainly on post-mortem connections for now.

The monitor overhead can be determined either for each event or for a complete execution. Since one type of event usually requires almost the same monitoring functionality during each execution, the overhead of an event can be

measured independently from the target program. To determine the overhead, we define the time of a generic communication event as follows:

$$t_{event} = t_{startup} + d_{msg} \cdot t_{transfer}$$

In this case, $t_{startup}$ is the necessary time to prepare the message transfer, while d_{msg} is the number of elements (usually bytes) in the message and $t_{transfer}$ is the amount of time necessary to transfer one element. The main difference between these two times is, that $t_{startup}$ will always delay the process, while $t_{transfer}$ may occur in the background. Additionally, there may be some blocking time, which is not relevant for determining the overhead.

When executing the program with the monitor, the time t_{event} is increased by the functionality performed in the monitor's init and exit parts. Obviously, most of the monitoring functionality will delay the process and is therefore added to the startup-time. Furthermore, the monitor will affect the transfer-time, for example, if a time-stamp or a vector-clock has to be transferred together with the message. Thus, the time for each event during monitoring can be computed as follows:

$$t_{event} = t_{startup} + t_{mon} + (d_{msg} + d_{mon}) \cdot t_{transfer}$$

Here t_{mon} is the amount of time spent during the monitor's activities, while d_{mon} is the amount of monitoring data added to the transferred message. Thus, the monitor overhead per event can be defined as follows:

$$t_{overhead} = t_{mon} + d_{mon} \cdot t_{transfer}$$

Computing the complete overhead of a process is then based on the overhead accumulated from all events occurring on that particular process during the program's execution. Clearly, this overhead depends on the implementation, the computation path selected during the execution, and the underlying hardware architecture.

Defining the overhead for a complete program is more difficult. Firstly, there is the question about how to combine the overheads achieved on all processes. Adding all process overheads is one possibility, but it neglects the fact that processes (and therefore monitoring functionality) is executed concurrently. On the other hand, the processes' overheads can be combined by computing minimum, average, and maximum values, which may fit the users' requirements better.

Secondly, there are some problems, which may invalidate all the overhead computation and prohibits a useful comparison between the program's execution with and without the monitor. One problem is, that the overhead may overlap with waiting times, and may therefore be not critical at all. Another problem is associated with network traffic, where it is possible, that less network contention is achieved if the program is running with monitor. This could even mean, that an instrumented program may execute even faster than it's corresponding original program.

4 Benchmarking with SKaMPI

To provide a standard method for evaluating and comparing different monitor approaches, we applied SKaMPI [9], the Special Karlsruher MPI-Benchmark. SKaMPI measures the performance of MPI [10] implementations and of the underlying hardware. Its primary goal is to support software developers, since knowledge of MPI function’s performance has several benefits, such as: shorter performance tuning after programming, consideration of performance issues of *all* targeted platforms during the design stage (what leads to increased portability), and program development independently of the target platform.

Basically, there are two ways to use SKaMPI for new measurements:

- (a) *Instrumenting an applications with SKaMPI functionality.* For this purpose we developed the library SKaLib. In this library we packaged the mechanisms of SKaMPI, such as precision adjustable measurement of time, controlled standard error, automatic parameter refinement, and merging results of several benchmarking runs. The advantage of this approach is, that SKaMPI mechanisms can be reused as needed. In addition, SKaLib is usable when developing sequential benchmarks, since it does not require MPI.
- (b) *Enhancing SKaMPI.* It is easy to add new functions to be measured to SKaMPI. Choosing this way means, that the whole SKaMPI infrastructure could be reused, which proved useful in our case. In principle, our approach is as follows. First, we measure the function of MPI without applying a monitor. Secondly, we instrument the MPI code to insert the monitor into the program and repeat the measurements. Afterwards the overhead of the monitor can be evaluated by comparing both results.

The remaining question is concerned with the kind of required measurements. Clearly, this depends on the statements that are instrumented. Corresponding to the definitions in section 3, we can identify two different kinds of tests. On the one hand, we have to evaluate the delay occurring on each process, which we called t_{mon} before. This can easily be achieved by measuring the time required for a function call to a communication function.

On the other hand, we require the increased transfer times, which are defined by d_{mon} . Contrary to the delay on one process, there are two processes associated with this problem. Therefore, we have to perform ping-pong measurements, where one process sends a message to its communication partner, which returns the message immediately. The communication time is then computed by removing the on-process delays for a send and receive function and dividing the remaining time by two, because message transfer occurs twice.

5 Example and Results

This section provides first experimental results of determining the results for a particular monitor system. The system under consideration is the record & replay module NOPE of the Monitoring And Debugging environment MAD [4],

which is a toolset for performance analysis and error detection in message passing parallel programs. Since the measurements of NOPE's overhead also depend on the implementation of MPI on the underlying operating system and hardware architecture, we performed our measurements on the SGI/Cray Origin 2000 with its native MPI implementation and the current version of the Cray Message Passing Toolkit.

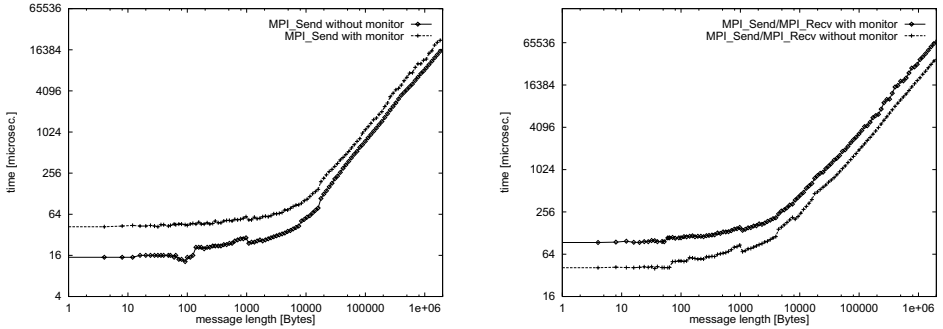


Fig. 2. SKaMPI measurements of: monitor overhead at `MPI_Send` (left) and monitor overhead for ping-pong communication (right)

The total overhead of a monitor depends on the overhead of the monitor at each of the communication functions. To determine this overhead for an example, we selected the function `MPI_Send` as displayed in the left graph of Figure 2. The horizontal axis displays the number of bytes that have been sent, while the vertical axis displays the time required for that particular function call. The two curves display the times of `MPI_Send`, once without any observation, once with the monitor NOPE applied. Consequently, the overhead of each function is the difference between both curves.

Additionally, we determined the overhead during message transfer. As mentioned before, this can be attributed to the additional data being required for monitoring purposes. The applied measurements are ping-pongs between two corresponding `MPI_Send` and `MPI_Recv` function calls. The results are displayed in diagram of Figure 2. Again, the upper curve shows the ping-pong communication with the monitor overhead, while the lower curve represents the original performance. Please note, that after removing the on-process delays from the curves of the right graph of Figure 2, we found out, that the monitor overhead established during message transfer can be neglected. This stems from the fact, that the target monitor implementation applies Lamport clocks [5], and only one additional byte has to be transferred.

6 Conclusion and Future Work

The importance of measuring the monitor overhead is well-known in the tool developers domain, but has either been ignored or not been defined as much as it might be required in some cases. With SKaMPI it is possible to perform

standardized benchmarking of MPI monitors and compare this measurements to obtain a characteristic criterion for the quality of different monitor implementations. As a side-effect, this measurements increase the user's awareness of the monitor overhead and it's implications for program analysis.

The future work in this project covers two main issues: On the one hand, the presented features of SKaMPI will be integrated and made available for other developers to measure their tools. Only by measuring several tools a comparison between different approaches is possible. Another interesting idea would be to measure one monitor approach available for different platforms, like for example tools based on the On-line Monitoring Interface Standard OMIS [11].

On the other hand, we will use the results of SKaMPI in a utility for monitor overhead removal, which covers timing delays of event occurrences and event reordering. So far, our work in this area has been performed with stand-alone measurements, which were rather difficult to adapt to the variety of available systems. Therefore it would be a big improvement in terms of standardization, if SKaMPI and it's monitor overhead measurements can be used in the scope of this project.

Remarks Public versions and further details of SKaMPI are available at:
<http://www.wipd.ira.uka.de/~skampi>

References

1. J. Cargille, B.P. Miller: Binary Wrapping: A Technique for Instrumenting Object Code. SIGPLAN Notices, Vol. 27, pp. 17–18 (June 1992).
2. J. Gait: A Probe Effect in Concurrent Programs. Software - Practise and Experience, Vol. 16(23), pp. 225–233 (March 1986).
3. D. Kranzlmüller, S. Grabner, J. Volkert: Monitoring Strategies for Hypercube Systems. Proc. PDP'96, 4th EUROMICRO Workshop on Parallel and Distr. Proc., Braga, Portugal, pp. 486–492 (Jan. 1996).
4. D. Kranzlmüller, S. Grabner, J. Volkert: Debugging with the MAD Environment. Parallel Computing, Vol. 23, No. 1-2, pp. 199–217 (Apr. 1997).
5. L. Lamport: Time, Clocks, and the Ordering of Events in a Distributed System. Comm. ACM, pp. 558–565 (July 1978).
6. E. Leu, A. Schiper, A. Zramdini: Execution Replay on Distributed Memory Architectures. Proc. 2nd IEEE Symp. on Parallel & Distributed Processing, Dallas, TX, pp. 106–112 (Dec. 1990).
7. A. Malony, D. Reed: Models for performance perturbation analysis. Proc. Workshop Parallel Distributed Debugging, ACM SIGPLAN/SIGOPS and Office of Naval Research, (May 1991).
8. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard - Version 1.1. <http://www.mcs.anl.gov/mpi/> (June 1995).
9. R. Reussner, P. Sanders, L. Prechelt, M. Müller: SKaMPI: A Detailed, Accurate MPI Benchmark. Proc. 5th European PVM/MPI Users' Group Meeting, Springer, Lecture Notes in Computer Science, Vol. 1497, Liverpool, UK, pp. 52–59 (Sept. 1998).
10. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra: MPI - The Complete Reference. 2nd Edition, MIT Press, Cambridge, Massachusetts, (1998).
11. R. Wismüller: On-Line Monitoring Support in PVM and MPI. Proc. of EuroPVM/MPI'98, LNCS, Springer, Vol. 1497, Liverpool, UK, pp. 312–319, (Sept 1998).

A Standard Interface for Debugger Access to Message Queue Information in MPI

James Cownie¹ and William Gropp² *

¹ Etnus Inc., Framingham MA, USA
jcownie@etnus.com,

WWW home page: <http://www.etnus.com/>

² Mathematics and Computer Science Division, Argonne National Laboratory
gropp@mcs.anl.gov,

WWW home page: <http://www.mcs.anl.gov/~gropp>

Abstract. This paper discusses the design and implementation of an interface that allows a debugger to obtain the information necessary to display the contents of the MPI message queues. The design has been implemented in the TotalView debugger, and dynamic libraries that conform to the interface exist for MPICH, as well as the proprietary MPI implementations from Compaq, IBM, and SGI.

1 Introduction

In any debugging task one of the debugger's main objectives is to make visible information about the state of the executing program.

Debuggers for sequential processes provide easy access to the state of the process, allowing the user to observe the values of variables, machine registers, stack backtraces, and so on. When debugging message-passing programs all of this information is still required; however, additional information specific to the message passing model is also needed if the user is to be able to debug the problems, such as deadlock that are specific to this new environment. Unfortunately the message-passing state of the program is not easily accessible to the user (or the debugger), because it is represented by data structures in the message-passing library, whose format and content are implementation dependent.

To address this problem, we describe an interface within the debugger itself that allows library-specific code to extract information describing the *conceptual* message-passing state of the program so that this can be displayed to the user.

2 The User's Model of MPI Internal State

Since each MPI communicator represents an independent communication space, the highest level of the user's model is the communicator. Within a communi-

* This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

cator there are, conceptually at least, three distinct “message queues,” which represent the state of the MPI subsystem. These are

Send Queue: Represents all of the outstanding send operations.

Receive Queue: Represents all of the outstanding receive operations.

Unexpected Message Queue: Represents messages that have been sent to this process, but have not yet been received.

The send and receive queues store information about all of the unfinished send and receive operations that the process has started within the communicator. These might result either from blocking operations such as `MPI_Send` and `MPI_Recv` or nonblocking operations such as `MPI_Isend` or `MPI_Irecv`. Each entry on one of these queues contains the information that was passed to the function call that initiated the operation. Nonblocking operations will remain on these queues until they have completed and have been collected by a suitable `MPI_Wait`, `MPI_Test`, or one of the related multiple completion routines. The unexpected message queue represents a different class of information, since the elements on this queue have been created by MPI calls in other processes. Therefore, less information is available about these elements. In all cases the order of the queues represents the order that the MPI subsystem will perform matching (this is important where many entries could match, for instance when wild-card tag or source is used in a receive operation).

Note that these queues are *conceptual*; they are a description of how a user can think about the progression of messages through an MPI program. As we discuss below, an MPI implementation may have fewer or more queues, and the queues within an implementation may be different. The interface described in this paper addresses how to extract these *conceptual* queues from the implementation so that they can be presented to the user independently of the particular MPI implementation.

3 Queues in Actual MPI Implementations

The MPICH implementation of MPI [1] maintains only two queues in each process: a queue of posted receives and a queue of unexpected messages. There is no explicit queue of send operations; instead, all of the information about an incomplete send operation (e.g., an `MPI_Isend`) is maintained in the associated `MPI_Request`. In addition, operations associated with all communicators live in the same queues, being distinguished internally by a context identifier.

Other organizations of message queues are possible. For example, Morimoto et al [2] describe a system where posted receives from a particular source are sent to the processor that is expected to send the message; in this implementation there is a queue of unmatched receives that send operations try to match before sending data to a destination. This does not conflict with the queue model that we have described; it is always possible for an MPI implementation to present the three queues that this interface requires. However, this internal queue of “expected receivers” is not visible to the user via the debug interface described here.

4 How to Extract State for a Debugger

While it would be possible to define a set of data structures that an MPI library must implement solely so that a debugger could extract the data it needed, specifying such implementation details was at conflict with the aim of the MPI Forum of defining an interface that would allow implementations to achieve low latency. Therefore, although we considered the idea of making such a proposal to the MPI-2 Forum, we decided against it, since it would clearly never be accepted.

Another apparently attractive solution would be to provide MPI calls that could be made inside a process to return information about the MPI library state. Unfortunately, this approach has a significant problem: it requires that there be a thread in the process that the debugger can use at any time to call the inquiry function. In many cases of interest no such thread exists; for instance, when debugging core files, there are no threads at all, or when debugging a deadlocked set of processes, all of the threads are likely to be in the system calls underlying the send or receive operations. These arguments lead to the conclusion that accessing the library state must be driven by the debugger, rather than code running inside the process.

One way to permit the debugger to display MPI library state would be explicitly to code the knowledge of the MPI library's internal data structures into the debugger. While this works (and indeed was the way that we initially implemented MPI queue display for MPICH in TotalView [3]), it is unattractive because it links the debugger and MPI library versions together inextricably and it requires that the writers of the debugger have access to the details of the MPI implementation. For vendor-specific MPI implementations (when the MPI vendor and debugger vendor are different), such access is hard or impossible to obtain (often requiring the involvement of lawyers).

4.1 Use of a Library Dynamically Linked into the Debugger

The solution that we have now adopted (and which is supported by TotalView) is to have the debugger use the Unix `dlopen` call to load a dynamic library (the “debug DLL”) at run time, when the debugger knows which MPI implementation is being used. This debug DLL is provided and distributed by the writers of the target MPI library.

This allows the debugger to be insulated from the internals of the MPI library, so that changes to the MPI implementation do not require the debugger to be rebuilt. Instead, the MPI implementation's specific debug DLL must be changed, but this is the responsibility of the MPI implementors. It allows the debugger to support multiple MPI implementations on the same system (for instance, both MPICH and a vendor MPI implementation) and to be portable, requiring no changes to its MPI support to work with many different MPI implementations. Finally, it allows implementors of MPI to provide their users with high-level debugging support without requiring access to the source code of the debugger.

5 The Interface between Debugger and Debug DLL

All calls to the debug DLL from the debugger are made through entry points whose names are known to the debugger. However, all calls back to the debugger from the debug DLL are made through a table of function pointers that is passed to the initialization entriypoint of the debug DLL. This procedure ensures that the debug DLL is independent of the specific debugger from which it is being called.

5.1 Opaque Objects Passed through the Interface

The debugger needs to identify to the debug DLL a number of different objects (mq_s for “message queue system”):

mq_s.image: An executable image file.

mq_s.process: A specific process.

mq_s.type: A named target type (**struct** or **typedef**).

However, the debugger does not want to expose its internal representations of these types to the debug DLL, which has no need to see the internal structure of these objects, but merely uses them as keys to identify objects of interest, or to be passed back to the debugger through a callback.

These objects are therefore defined in the interface file as **typedefs** of undefined structures and are always passed by reference. The use of these opaque types allows the debugger the freedom either to pass true pointers to its internal data structures or to pass some other key to the debug DLL from which it can later retrieve its internal object. We prefer **typedefs** of undefined structures rather than simply using **void ***, since using the **typedefs** provides more compile-time type checking over the interface.

For reasons of efficiency it is important that the debug DLL be able easily to associate information with some of these debugger-owned objects. For instance, it is convenient to extract information about the address at which a global variable of interest to the debug DLL lives only once for each process being debugged, rather than every time that the debug DLL needs access to the variable. Similarly, the offset of a field in a structure that the debug DLL needs to understand is constant within a specific executable image, and again should be looked up only once. Callbacks are therefore provided by the debugger to allow the debug DLL to store and retrieve information associated with image and process objects. Since retrieving the information is a callback, the debugger has the option either of extending its internal data structures to provide space for an additional pointer or of implementing a hash table to associate the information with the process key.

5.2 Concrete Objects Passed through the Interface

To allow the debugger to obtain useful information from the debug DLL, concrete types are defined to describe a communicator and a specific element on a message queue.

The information in the `mqs_communicator` structure includes the communicator's size, the local rank of the process within the communicator, and the name of the communicator as defined by the MPI implementation or set by the user using the MPI-2 [4] function `MPI_Comm_set_name`, which was added to the standard specifically to aid debugging and profiling.

The `mqs_pending_operation` structure contains enough information to allow the debugger to provide the user with details both of the arguments to a receive and of the incoming message that matched it. All references to other processes are available in the `mqs_pending_operation` structure both as indices into the group associated with the communicator and as indices into `MPI_COMM_WORLD`. This avoids any need for the debugger to concern itself explicitly with this mapping.

5.3 Target Independence

Since the code in the debug DLL is running inside the debugger, it could be running on a completely different machine from the debugged process. Therefore, the interface uses explicit types to describe target types, rather than using canonical C types. The interface header definition file defines the types `mqs_taddr_t` and `mqs_tword_t` that are appropriate types for the debug DLL to use on the host to hold, respectively, a target address (`void *`) and a target word (`long`).

It is also possible that although the debugger is running locally on the same machine as the target process, the target process may have different properties from the debugger. For instance, on AIX, IRIX, or Solaris 7, it is possible to execute both 32- and 64-bit processes. To handle this situation, the debugger provides a callback that returns type size information for a specific process. To handle the possibility that the byte ordering may be different between the debug host and the target, the debugger provides a callback to perform any necessary byte reordering when viewing the target store as an object of a specific size.

The debugger also provides callbacks to the debug DLL to allow it to find the address of a global symbol, to look up a named type, to find the size of a type, and to find the offset of a field within a (`struct`) type. Each of these calls takes as an argument a specific process.

These callbacks enable the debug DLL to be entirely independent of the target process, as is demonstrated by the fact that the MPICH implementation of the debug DLL contains no target-specific `#ifdefs` yet is successfully used on a variety of both big- and little-endian 32- and 64-bit systems.

5.4 Accessing Target Process' Memory

As well as the services already described for extracting symbols and types and for supporting target independence, the debugger provides the debug DLL with a function for reading the store of a target process. This is the most fundamental service provided, since without it the debug DLL would have no access to target runtime state information.

6 Extracting the Information

The debug DLL functions called from the debugger to extract the information are structured as iterators: one to iterate over all of the communicators in a process, and a second to iterate over all of the messages in a specific message queue. This avoids the problems of store allocation that would arise were the interface defined in terms of arrays or lists of objects.

Since a communicator and associated translation groups can be of significant size (in MPICH a communicator and its associated groups occupy at least 176 bytes on a 32-bit machine), reading all of the communicator information each time that a message queue is displayed could be slow. Therefore a call to the routine `mqs_update_communicator_list` is made before the communicator iterator is initialized if the debugger knows that the process has run. This allows the DLL to hold the communicator information locally and to update it only when the communicator list has been changed. In the MPICH library such changes are detected by the use of a sequence number that is incremented whenever a communicator is created, destroyed, or modified (e.g., has its name changed).

To extract the communicator list, the debugger iterates over all of the communicators in the process using code similar to the following C++ example taken directly from TotalView. The MPI debug support library has been encapsulated in the C++ `dll` object, whose methods are trivial wrappers for the functions in the debug DLL itself.

```
/* Iterate over each communicator displaying the messages */
mqs_communicator comm;

for (dll->setup_communicator_iterator (process);
     dll->get_communicator (process, &comm) == mqs_ok;
     dll->next_communicator(process))
{
    /* Do something on each communicator, described by comm */
}
```

To extract information about a specific message queue in the currently selected communicator, code like this (again taken with only trivial edits from TotalView) is used.

```
int errcode = dll->setup_operation_iterator (process, which_queue);

if (errcode != mqs_ok)
    return false; /* Nothing to be done */

mqs_pending_operation op;
while (dll->next_operation (process, &op) == mqs_ok)
{
    /* Display the information about the operation from op */
}
```

7 Implementation of the Debug DLL for MPICH

To provide a send queue for the debugger, we added a small amount of code to the MPICH implementation to maintain a list of `MPI_Request`s associated with MPI Send operations; completion of these operations (with `MPI_Test`, `MPI_Wait`, or the related multiple-completion routines) removes the request from the list. This extra list is conditionally compiled into MPICH when MPICH is configured with the `--enable-debug` switch. The run-time cost of this change is minimal when the process is not being debugged, amounting to one additional conditional branch in each of the immediate send operations, and in each of the test or wait operations when completing a request created by an immediate send.

The rest of the implementation of the debug DLL uses existing MPICH data structures; changes in MPICH internals are invisible to the debugger. For example, a change in `MPI_Request` to handle requests freed by the user before the operation completed only required rebuilding the DLL (along with the MPICH library); this change appeared in the next release of MPICH without requiring any change in TotalView.

8 Use of the Message Queue Data

Message queue data extracted from an MPI program can be used in many ways. The most basic is just to display it to the user, as in Fig. 1.

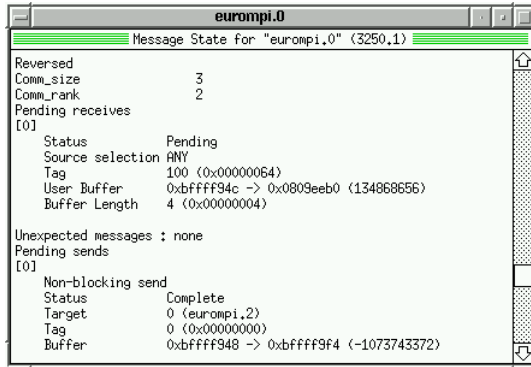


Fig. 1. Sample display generated by TotalView using the interface with MPICH

The data can also be used as input to tools that give higher-level information; for instance, Chan [5] describes a “Wait state graph” tool that uses the information extracted from an MPI implementation through this interface to show the user the message dependencies in an MPI program.

9 Availability of Implementations

In addition to the portable implementation that supports MPICH, implementations of libraries that conform to this interface have been written to support the proprietary MPI implementations of Compaq, IBM, and SGI.

The source code for the interface header and the MPICH implementation of the debug DLL are distributed with the MPICH source code, available from <http://www.mcs.anl.gov/mpi/mpich>. These files are open source; all rights are granted to everyone.

The code contains descriptions of other functions that have not been discussed here for lack of space, including conversion of error codes to strings and support for checking that the DLL and the client debugger are implementing compatible versions of the interface.

10 Conclusions

The interface described has proved to be useful on many systems. It successfully separates an MPI enabled debugger from the details of the MPI implementation and provides sufficient hooks to allow the debug DLL also to be written in a portable manner.

While the interface will require extensions to handle some of the features of MPI-2 (for instance, an index into `MPI_COMM_WORLD` will no longer be sufficient to identify a process), and other useful extensions such as the ability to display MPI data types, groups, or MPI-2 remote memory access windows are possible, the functionality already provided is extremely useful.

The interface is publicly available, and we encourage both debugger and MPI implementors to fetch the files and use the interface.

References

1. William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the MPI Message Passing Interface standard. *Parallel Computing*, 22(6):789–828, 1996.
2. Kenjii Morimoto, Takashi Matsumoto, and Kei Hiraki. Implementing MPI with the memory-based communication facilities on the SSS-CORE operating system. In Vassil Alexandrov and Jack Dongarra, editors, *Recent advances in Parallel Virtual Machine and Message Passing Interface*, volume 1497 of *Lecture Notes in Computer Science*, pages 223–230. Springer, 1998. 5th European PVM/MPI Users' Group Meeting.
3. Etnus, Inc. TotalView User's Manual. Available from <http://www.etnus.com>.
4. Message Passing Interface Forum. MPI2: A Message Passing Interface standard. *International Journal of High Performance Computing Applications*, 12(1–2):1–299, 1998.
5. Bor Chan. <http://www.llnl.gov/sccd/lc/DEG/TV-LLNL-Rel-3.html#waittree>.

Towards Portable Runtime Support for Irregular and Out-of-Core Computations

Marian Bubak^{1,2} and Piotr Łuszczek¹

¹ Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland

² Academic Computer Centre – CYFRONET, Nawojki 11, 30-950 Kraków, Poland

email: {bubak, luszczek}@uci.agh.edu.pl

phone: (+48 12) 617 39 64, *fax:* (+48 12) 633 80 54

Abstract. In this paper, we present the `lip` – a runtime system which enables easy and portable parallelization of irregular and out-of-core computations. Functions for handling irregular data were developed using the same concept as in the CHAOS library. Out-of-core parallelization is based on the idea of *in-core section*, and functions for out-of-core data are implemented with capabilities provided by MPI-IO. The new library may be used in C, Fortran and Java programs. Results of performance tests for a generic irregular out-of-core program on HP S2000 are presented and possible further extensions are discussed.

1 Introduction

Parallelization of so called *irregular* and *out-of-core* (OOC) applications is a very important issue in development of large scale computations [1]. The characteristic feature of *irregular problems* is accessing data through one or more levels of indirection arrays. During recent decade few runtime tools have been developed to facilitate parallelization of irregular problems, with the CHAOS runtime library [2] being the most notable among them. Recently, it has been successfully ported from Intel Paragon's NX to MPI [3]. Many applications involve primary data structures which are too large to fit into the main memory; they are called *out-of-core problems*. To parallelize this kind of programs one may use explicit I/O calls, e.g., those offered by the MPI-IO. There are also dedicated tools, e.g., PIOUS [4] and Panda [5] systems. Problems which are at the same time the irregular and out-of-core ones are especially difficult to parallelize. Typical examples include particle codes, CFD programs, sparse matrix computations. They were examined by a number of research projects what resulted in the development of runtime systems like VIPIOS [6] and COMPASSION [7].

In this paper, we present the `lip` – a portable runtime support library for both in- and out-of-core irregular problems. Portability is guaranteed by the use of the MPI with its I/O features ([8]). Parallelization of irregular problems is realized in the similar way as in the CHAOS library [2]. The out-of-core functions are based on the idea of *in-core section*. The functions of the `lip` may be used in C, Fortran and Java parallel programs developed with the MPI, as well as by the parallelizing compilers.

2 MPI-Based Support for Irregular Problems

Since originally the CHAOS library was implemented on top of the NX message passing system, the techniques described in [9] were used to convert basic communication routine calls to MPI along with additional improvements to the CHAOS [3] itself (Fig.1). The version of CHAOS for MPI programs, though fully operative, lacked several important features, namely, there was no support for computations on heterogeneous systems and inherently collective operations were implemented with simple `send()` and `recv()` functions. In the second stage, we have elaborated a new `lip` library which resolves all the problems mentioned above. It uses the higher-level collective MPI routines which improve its efficiency and strictly typed communication buffers to provide support for heterogeneous environments. API of the `lip` is very similar to that of the CHAOS.

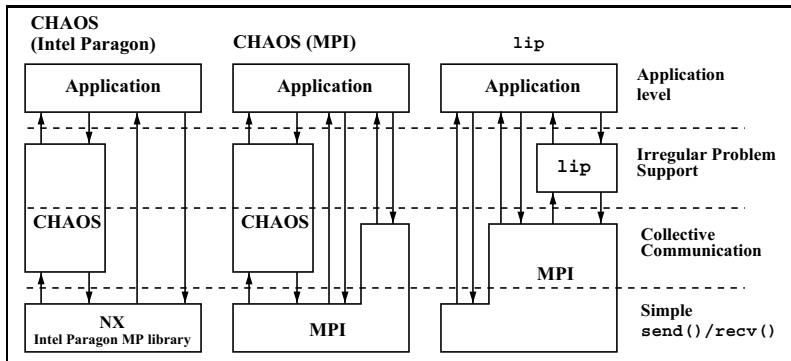


Fig. 1. Consecutive rearrangements of the CHAOS internal structure and the message passing usage.

There are at least three models of disk storage [10]: local placement model, global placement model, and partitioned in-core model. An alternative to providing I/O model is to use highly expressive I/O API calls which allow the user to fully describe the desired I/O operation. The `lip` aims at the second solution and allows the user to take advantage of the capabilities provided by the MPI-I/O. The I/O operations are performed by the user whereas the `lip` provides only the data structures which might be used in these operations.

3 Parallelization of OOC Problems

Runtime support for parallel execution of irregular code is based on so called *Inspector/Executor* technique introduced in the CHAOS [2]. It splits the whole process into two independent phases: *Inspector Phase*, *Executor Phase*.

In the *Inspector Phase* phase, data references (from indirection array) are examined and translated either into references to node's local data array (global

to local index translation) or into references to node's temporary buffer (referred to as a *ghost area*). The latter case is tightly connected with generation of *schedule* which is a data structure describing how communication is to be performed efficiently to bring the desired data into *ghost area*. The *Inspector Phase* is depicted in Fig. 2 as a transition from global to local view. The last part of the Fig. 2 shows OOC local view which pertains to the case when index array (which is examined during *Inspector phase*) is too large to fit into memory.

In the *Executor Phase* the results of *Inspector phase* – *schedule* – are used to perform communication (that brings non-local data into *ghost area* and is done through the call to the `LIP_Gather()` routine) and then computation is performed which uses the translated indices and local data array with an additional buffer space (*ghost area*). Next, another communication has to be performed to scatter results of the computation. The communication *schedule* is used again through a call to the `LIP_Scatter()` function.

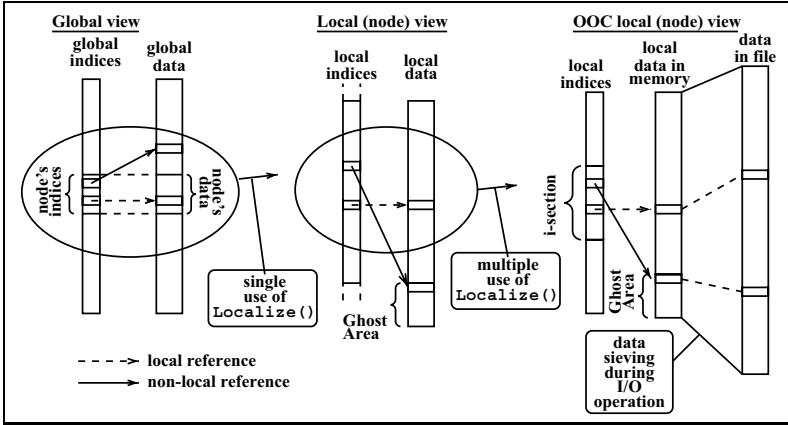


Fig. 2. Restructuring of arrays for OOC computing.

The *Inspector/Executor* approach remains almost unaltered due to introduction of the concept of the *in-core section* (tile or slab) [11]. The whole index array is stored in a file and only a small chunk of it (*i-section*) is loaded into the main memory to enable the optimization described previously. In this way, the technique successfully used for the irregular in-core problems have been generalized for the irregular OOC problems with *in-core* data arrays and OOC index arrays. A generic program which uses *i-section* is given in Fig. 3.

However, this concept has to be further modified when not only the index arrays but also the data arrays become *out-of-core*. The previously described *Inspector phase* is insufficient because only part of the data array is stored in the memory and it is possible for the index array to refer to the data element which is currently stored on a disk. The detailed analysis leads to a conclusion that three types of non-local accesses may be encountered in the index array: access


```

MPI_Init( /* ... */ );
LIP_Setup( /* ... */ );

/* Generate index array describing
 * relationships between user data */

/* for each i-section */
for ( /* ... */ )
{
    /* generate indices, e.g. mesh
     * connections */
    /* ... */

    /* write i-section's indices
     * to file */
}

/* Perform irregular computation
 * on the data */
for ( /* ... */ )
{
    /* read i-section's indices
     * from a file */

    /* Inspector Phase (computes
     * optimized communication pattern) */
    LIP_Localize( /* ... */ );

    /* Executor Phase (performs
     * communication and computation) */

    /* gather non-local irregular data */
    LIP_Gather( /* ... */ );

    /* perform computation on data */
    for ( i = /* ... */ )
    {
        k = edge[i];
        y[k] = f( x[k] );
    }

    /* scatterer non-local
     * irregular data (results) */
    LIP_Scatter( /* ... */ );
}

LIP_Exit( /* ... */ );
MPI_Finalize( /* ... */ );

```

Fig. 3. Sample C code which shows OOC computing with the lip.

to data of other node (the data are present in the node's main memory), access to data of other node (the data are *not* present in the node's main memory), access to local node data which are not present in memory (they are stored on a disk).

It would be heavily inefficient to read each element from a disk just as it becomes needed. Therefore, at the *Inspector Phase*, the optimization has to be aimed at efficient communication and also the I/O accesses associated with the *not-in-memory* data elements have to be minimized. Furthermore, the data transfer should be performed using the available optimization techniques such as: data sieving and collective I/O with MPI derived data types [13, 12].

We have used ROMIO 1.0.1 [14] (the portable implementation of the MPI-IO) which is included in the HP MPI 1.4 – MPI implementation from HP (built on the mpich [15]), which, in turn, was used in the tests we have performed (see section 5). ROMIO lacks some routines defined in the MPI-IO standard, namely the split collective routines that may be used to employ the performance-increasing technique of overlapping I/O operations with communication and computation.

4 Bindings to Fortran and Java

Since text lip is supposed to serve as a portable runtime support library not only across different platforms but also across different host languages, we have developed bindings of the library to Fortran and Java.

The Fortran wrappers could have been created using techniques presented already in [15, 16] and this could be done in a virtually automatic manner. How-

<pre> int main(int argc, char **argv) { /* ... */ /* initialize MPI * library */ MPI_Init(&argc, &argv); /* initialize LIP * library */ LIP_Setup(/...*/); /* Perform computation * using LIP and MPI */ LIP_Localize(/...*/); /* ... */ LIP_Gather(/...*/); /* ... */ LIP_Scatter(/...*/); /* ... */ /* leave LIP library */ LIP_Exit(); /* leave MPI library */ MPI_Finalize(); } </pre>	<pre> program main ! ... ! initialize MPI ! library call MPI_INIT(ierr) ! initialize LIP ! library call LIP_SETUP(...) ! Perform computation using ! LIP and MPI call LIP_LOCALIZE(...) ! ... call LIP_GATHER(...) ! ... call LIP_SCATTER(...) ! ... ! leave LIP library call LIP_EXIT(ierr) ! leave MPI library call MPI_FINALIZE(ierr) stop end </pre>	<pre> class Test { public static void main(String args[]) { // ... // initialize MPI // library MPI.Init(args); // initialize LIP // library LIP.Setup(); // Perform computation // using LIP and MPI LIP.Localize(/...*/); // ... LIP.Gather(/...*/); // ... LIP.Scatter(/...*/); // ... // leave LIP library LIP.Exit(); // leave MPI library MPI.Finalize(); } </pre>
--	--	--

Fig. 4. Calling the lip from C, Fortran and Java.

ever, we had to cope with the previously unresolved technical issue concerning translation of array indices (which are ubiquitously used in irregular applications) between C and Fortran array-indexing schemes. A much greater effort has to be invested to produce Java wrappers since MPI Forum does not legitimize any standard MPI to Java binding. This leads to the necessity of a much more careful choice of implementation details that would successfully resolve the platform-specific problems with *Java Native Interface*, which are mostly related to the handling of the shared libraries on different operating systems and seamless cooperation of these libraries not only with Java Virtual Machine but also with each other. Moreover, there is an issue related to the currently being investigated possibility of various extensions to Java language and its API for more efficient scientific computing (see e.g. [17]). Nevertheless, we have implemented an experimental Java binding for the lip with mpich [15] since the LAM implementation of MPI [18] encountered problems with Java Virtual Machine that fork()'ed. The work on other implementations of wrappers is in progress. Fig. 4 presents generic programs in C, Fortran and Java with calls to the lip functions.

5 Sample Results

To evaluate performance of the `lip` we have used generic irregular loop which operates on large data and index sets (see Fig. 3). First, data are generated along with indices (which could describe the mesh or graph connections), and next, they are used to index the data array. The index array is OOC, so it is divided into *i-sections* and stored on disk(s). During the computation phase, the *i-sections* are loaded and used.

#processors	1	2	4	6	8	10	12	14
Elapsed time (in seconds) for local I/O.								
NOOC small	204.8	111.2	60.3	42.6	34.2	29.4	26.5	25.1
OOC small	204.3	110.7	59.6	43.3	34.7	30.2	27.8	25.8
OOC big	273.2	166.0	100.4	78.8	66.6	63.1	53.6	56.4
Elapsed time (in seconds) for global I/O.								
NOOC small	204.8	111.2	60.3	42.6	34.2	29.4	26.5	25.1
OOC small	204.0	111.3	61.2	44.2	35.9	32.1	29.6	27.9
OOC big	266.5	168.0	116.1	90.7	82.9	77.4	73.6	73.6
Elapsed time (in seconds) for constant-time problem.								
Local I/O model	188.1	205.1	215.2	233.0	244.3	257.2	269.8	289.7
Global I/O model	188.5	204.4	223.5	236.7	256.1	276.2	297.9	401.2

Table 1. The timings for execution of a simple irregular loop.

Three test cases were run. The first one (*NOOC small*) creates data (of global size 23063040 C language `double` values) and index arrays (of global size 5765760 C language `int` values) which are large enough to fill up all the memory that a user process can allocate. In the second case (*OOC small*), the sizes of the arrays are the same as previously but the `lip` handles I/O accesses (instead of the Exemplar’s virtual memory system) through the MPI-IO (*i-section* size is 360360). In the third case (*OOC big*), the arrays are so large (data: 2882880; indices: 57657600; *i-section*: 360360) that they can only be handled through the `lip` (process cannot allocate sufficient memory to fit them into the main memory even with the virtual memory system support).

The tests were run on the HP S2000, on a dedicated 14-processor subcomplex. The computer itself has one 16-processor hypernode with 4 GB memory, 6 physical disks (of total size 49 GB) connected through 3 PCI interfaces. One of the disks (4 GB alone on one PCI interface) serves as a swap space for the virtual memory management system whereas the remaining 5 disks (9 GB each) are for the user (they are connected through 2 remaining PCI interfaces - 2 disks on the first one and another 3 on the second one).

The timings for different runs are shown in Table 1. The first two *small* cases show virtually the same performance of the `lip` and the virtual memory system. The *OOC big* case with its speedup below 5 for 14 MPI processes reveals the weakness of the tested system, which stems from the fact that it has only

five disk drives. Overlapping of I/O and computation would hide this flaw to some extent, unfortunately, the MPI-IO implementation we have used, have not appropriate capabilities (i.e. the full support for all kinds of non-blocking I/O routines). Table 1 shows also the elapsed time of application that had a constant data size for a single process. In this case, the performance of local an global I/O can be interpreted in favor of the former as it was also true for the previously described results. A possible explanation is an inefficient implementation of collective routines (again, ROMIO 1.0.1 lacks some of them), since we did not cope with the Exemplar-specific system calls which may perform better.

6 Conclusions and Future Work

An important feature of the *lip* is its support for efficient handling of irregular access patterns (as they are encountered in irregular problems) through the high-level data structures called *schedules*. This is done in a uniform manner for the I/O access routines and for communication. With these data structures, information may be transferred from the phase of analysis of the user data (*Inspector Phase*) to the communication and I/O phase (*Executor Phase*). Then it is possible to take full advantage of the MPI-IO routines expressiveness.

Our goal was to provide the user with higher level routines than those available in the MPI-IO but still retain the flexibility of the MPI library so the user could further tune his program through direct calls to the MPI-IO. On the other hand, the *lip* routines perform as much optimization as possible (such as the data sieving described in [12, 13]) with the information gained while examining the user data. This may be useful when the user does not want to cope with MPI-IO routines directly. Besides the classical large scale computing the *lip* may also be applied to develop the interfaces for the high performance storage and retrieval of data [19] providing support for parallelism and out-of-core operations, on the other hand, owing to Java bindings, it may be a very important extension to HP Java [20].

The future work includes implementation of efficient data partitioning and remapping routines which should result in a better load-balance and improved efficiency. In addition to this, we are currently working on improvement of the Java binding to the *lip* in order to develop portable and robust wrappers.

Acknowledgments We would like to thank Prof. Peter Brezany for introducing us into the field of out-of-core problems and for many valuable discussions. We are also grateful to Mr Dawid Kurzyniec for his collaboration and to Ms Zofia Mosurska and Dr. Piotr Wyrostek from the ACC CYFRONET for their help. This research was done in the framework of the Polish-Austrian collaboration and it was supported in part by the KBN grant 8 T11C 006 15.

References

1. Brezany, P.: Input/Output Intensively Parallel Computing, LNCS **1220**, Springer, Berlin Heidelberg New York (1997).

2. Saltz, J., et al.: A Manual for the CHAOS Runtime Library, UMIACS Technical Reports CS-TR-3437 and UMIACS-TR-95-34, University of Maryland; March 1995; http://hpsl.cs.umd.edu/pub/chaos_distribution
3. Bubak, M., Łuszczyk, P., and Wierzbowska, A.: Porting CHAOS Library to MPI. In Alexandrov, V., Dongarra, J., (eds.) *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Proc. 5th European PVM/MPI Users' Group Meeting, Liverpool, UK, September 7-9, 1998, LNCS **1497**, Springer, pp. 131-136.
4. Moyer, S. and Sunderam, V.S.: Parallel I/O as a Parallel Application. *International Journal of Supercomputer Applications*, **9** (1995) 95-107.
5. Seamons, K. E., Chen, Y., Jones, P., Jozwiak, J., and Winslett, M.: Server Directed Collective I/O in Panda. In *Proc. Supercomputing'99, San Diego, California, December 1995*, <http://drl.cs.uiuc.edu/panda/>
6. Brezany, P., Mueck, T.A., and Schikuta, E.: A Software Architecture for Massively Parallel Input/Output. In Waśniewski, J., et al.: *Applied Parallel Computing, Proc. PARA'96*, Lyngby, August 1996, LNCS **1184**, Springer, pp. 85-96.
7. Carretro, J., No, J., Park, S., Choudhary, A., and Chen, P.: COMPASSION: A Parallel I/O Runtime System Including Chunking and Compression for Irregular Applications. In: Sloot, P., Bubak, M., Hertzberger, B., (eds.): *Proc. Int. Conf. High Performance Computing and Networking*, Amsterdam, April 21-23, 1998, LNCS **1401**, Springer, 1998, pp. 668-677.
8. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface, July 18, 1997; <http://www.mpi-forum.org/docs/mpi-20.ps>
9. Saphir, W.: Porting Applications from NX to MPI. Talk presented at NAS on July 26, 1994, <http://parallel.nas.nasa.gov/Parallel/Talks/nx2mpi/nx2mpi.html>
10. Bordawekar, R., Choudhary, A.: Issues in Compiling I/O Intensive Problems. In Jain, R., Werth, J., and Browne, J.C. (Eds.) *Input/Output in Parallel and Distributed Computer Systems*, chapter 3, pp. 69-96, Kluwer, 1996.
11. Brezany, P., Choudhary, A., and Dang, M.: Parallelization of Irregular Codes Including Out-of-Core Data and Index Arrays. In *Proceedings of Parallel Computing 1997 - PARCO'97*, Bonn, Germany, September 1997, Elsevier, pp. 132-140.
12. Thakur, R., Gropp, W., and Lusk, E.: Data Sieving and Collective I/O in ROMIO. In *Proc. of the 7th Symposium on the Frontiers of Massively Parallel Computation*, February 1999, pp. 182-189.
13. Thakur, R., Gropp, W., and Lusk, E.: A Case for Using MPI's Derived Datatypes to Improve I/O Performance. In *Proc. of SC98: High Performance Networking and Computing* November 1998.
14. Thakur, R., Lusk, E., and Gropp, W.: Users Guide for ROMIO: A High-Performance, Portable MPI-IO Implementation. *Technical Memorandum ANL/MCS-TM-234*, Mathematics and Computer Science Division, Argonne National Laboratory, Revised July 1998; <http://www.mcs.anl.gov/romio/>
15. MPICH – A Portable Implementation of MPI, <http://www.mcs.anl.gov/mpi/mpich/>
16. Geist, A., et al.: PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, Massachusetts (1994)
17. Java Grande Forum, <http://www.javagrande.org/>
18. LAM/MPI Parallel Computing: <http://www.mpi.nd.edu/lam/>
19. Brezany, P., Winslett, M.: Advanced Data Repository Support for Java Scientific Programming In: Sloot, P., Bubak, M., Hoekstra, A., Hertzberger, B., (eds.): *Proc. Int. Conf. High Performance Computing and Networking*, Amsterdam, April 12-14, 1999, LNCS **1593**, Springer, 1999, pp. 1127-1136.
20. HP Java Project, <http://www.npac.syr.edu/projects/pcrc/HPJava/>

Enhancing the Functionality of Performance Measurement Tools for Message Passing Environments

Marian Bubak^{1,2}, Włodzimierz Funika¹, Kamil Iskra¹, Radosław Maruszewski¹, and Roland Wismüller³

¹ Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Kraków, Poland

² Academic Computer Centre – CYFRONET, Nawojki 11, 30-950 Kraków, Poland

³ LRR-TUM – Technische Universität München, D-80290 München, Germany

{bubak, funika, iskra}@uci.agh.edu.pl, maruszew@icrsr.agh.edu.pl,
wismuell@in.tum.de

phone: (+48 12) 617 39 64, fax: (+48 12) 633 80 54

Abstract. The paper presents the concept of and insight into enhancing the portability and functionality of two performance measurement tools – PATOP and TATOO – in order to adapt them to work with message passing applications. We discuss the concepts of porting the tools to the OCM monitoring environment, the structure of the modified tools and the extensions made, as well as implementation issues.

1 Introduction

Performance measurement tools for message passing applications are intended to help in finding and fixing bottlenecks either via straightforward analysis of the state of application processes' execution and the communication between processes, or by modelling the location, time and cause of the bottleneck [1]. During recent years, a considerable number of performance tools have been made available. Unfortunately, as a rule off-line tools feature the following principal disadvantages: potentially vast volumes of performance data, poor guidance and insufficient flexibility. In turn, existing on-line tools are rather difficult to port to parallel environments which are implementations of the *de facto* standards, such as PVM [2] and MPI [3]. Moreover, making two or more on-line tools analyse or visualize the behaviour of the same application concurrently is virtually impossible.

Our efforts are aimed at development of performance measurement tools suitable for monitoring parallel applications that use the message passing paradigm, ensuring that the tools are portable to all current and future message passing libraries. Currently we are interested in monitoring PVM and MPI applications. In addition, we aim at enabling the interoperation of the tool with other program development support tools, like debuggers, visualisers etc., thus allowing to run such tools concurrently, while operating on the same instance of parallel application.

In order to realize these goals we decided to port and enhance the functionality of two existing performance measurement tools, PATOP [4] and TATOO [5], using the On-line Monitoring Interface Specification OMIS [6] and the OCM (OMIS Compliant Monitoring system) for low-level monitoring. The layer of abstraction provided by OMIS is used to gain the necessary flexibility, portability and interoperability,

In the subsequent sections we will discuss the functionality of the existing tools and the modifications made to support message passing programs. Preliminary concepts have been presented in [7].

2 PATOP and TATOO on Top of the OCM

PATOP and TATOO, two tools for on-line (PATOP) and off-line (TATOO) performance analysis, have originally been developed at LRR-TUM for the analysis of applications running under the PARIX environment on Parsytec systems. Performance visualisation is provided using an X11/Motif user interface. Due to almost identical user interfaces in both PATOP and TATOO, users do not have to learn twice how to use them. Both tools provide reasonably rich sets of available measurements and display diagrams. They support measurement and visualisation at various levels of detail: the whole system, individual nodes, particular processes or user functions. The available measurements comprise *busy* and *sojourn time*, *delay* in sending and receiving, *data volume* sent and received. Performance data can be presented in a wide range of ways, as bar graphs, multi-curves, color bars, curve/matrix graphs, distrigrams, and matrix diagrams.

OMIS [6] has been designed to define a standardised interface between run-time tools and the parallel programming environments, an interface which is independent of the parallel environment, while providing a universal, portable and extensible API. OMIS includes the definition of a basic set of monitoring services divided into three groups: *information* services, *manipulation* services and *event* services. The basic set can be augmented with extension services to support requests specific to a particular purpose or environment. The OCM is a reference implementation of OMIS, which provides support for monitoring message passing applications under PVM (a version for MPI is about to be issued). The OCM is a distributed monitoring environment, with a central Node Distribution Unit and local monitoring processes (one per node), communicating with the monitored processes via shared memory. It is worth mentioning that an alternative approach which provides a software architecture for the implementation of monitoring is presented in [8].

The integration of PATOP with the OCM involves an additional layer, ULIBS to ease interfacing PATOP to the OCM. The task of ULIBS is to transform high-level measurement specifications to event/action relations accepted by the OCM and to transform back the results. In Fig. 1 the structure of the on-line monitoring environment is presented with monitoring a sample parallel application. In order to enable the monitoring of a parallel application by the OCM, some parts of the OCM must belong to the target system. To provide run-time tools with

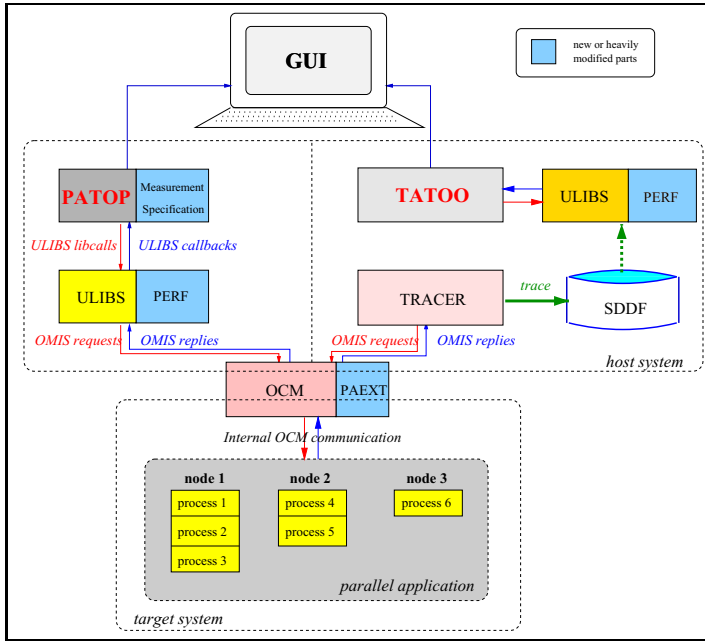


Fig. 1. Operation of PATOP and TATOO on top of the OCM

full functionality, such as the monitoring of communication library calls, parts of OCM must be linked to the application being monitored. The PATOP tool runs on a host machine and communicates with the OCM using the abstraction layer, ULIBS.

The structure of the TATOO tool is shown on the right side of Fig. 1. Performance data are stored in local trace buffers within the monitored application processes. When the application terminates, or a trace buffer is almost full, the OCM reads its contents. The trace data is then sent to TRACER – a standard OMIS client distributed with the OCM. TRACER saves the data to a trace file in the SDDF format. The TATOO tool reads this trace file, using a special, off-line version of ULIBS. Adapting PATOP and TATOO to PVM and MPI required the definition of a number of new services for performance analysis and a modification of the ULIBS functionality.

3 Gathering of Performance Data Using OCM

In order to be able to provide performance analysis of a parallel program, tools need to acquire information about the program such as the amount of time spent on waiting for a message from another process, or the message size. The OCM provides all the necessary support for obtaining such information. There are three principal ways of making this data available to the performance analysis tools:

1. Using *direct communication* — whenever an interesting event is detected, the OCM executes the services that have been associated with that event. The services send their results (i.e. information on the event or the state of the application when the event occurred) to the tool. Fig. 2 shows the flow of control and data in this case.
2. Using a *trace* — when a relevant event occurs, some information on the event is stored in a trace buffer local to the application process. It is later read by the OCM on request. The tracing mechanism is available via an OMIS extension included in the OCM. Fig. 3 shows the flow of control and data in this approach. It is also valid for the integrator/counter approach described below.
3. Using *counters* and *integrators* — the most important information on each event (e.g. the message length for a send event) is summarized in counters and integrators local to the application process and read by the OCM on request. The mechanism supporting the counters and integrators is part of PAEXT — a performance analysis extension to the OCM, which is outlined in Section 4.

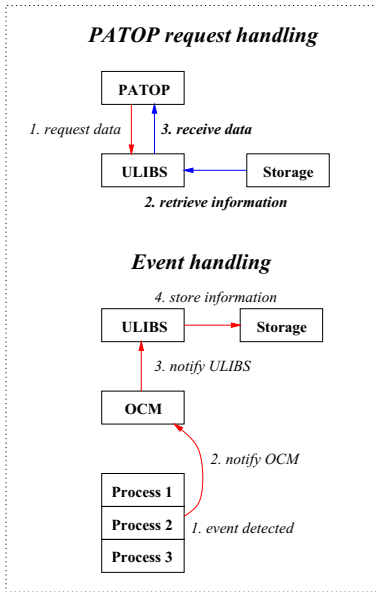


Fig. 2. Direct communication mechanism

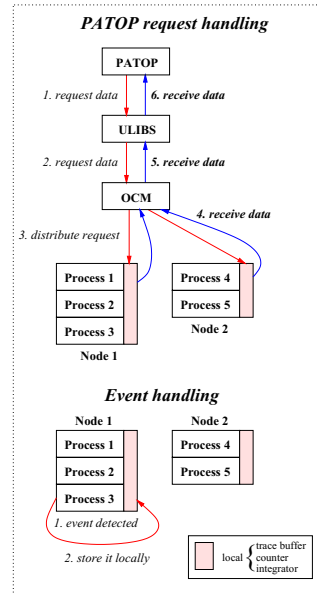


Fig. 3. Tracing and counting/integrating mechanisms

When choosing the proper way of gathering performance data it is indispensable to take into consideration the demand for a particular kind of performance data and the cost of gaining it. The *direct communication* allows to acquire not

only the full information on each event, but also information on the state of the application when the event occurred. However, it also has the greatest cost, as a message needs to be sent for each event occurrence. In the *tracing* and *counter/integrator* approaches, communication is needed only infrequently when the displays of PATOP are updated. On the other hand, less information can be retrieved. While *tracing* at least allows to store detailed information on each single event, *counters and integrators* only provide summary information. But of course, the data volume that needs to be transferred is therefore minimal. Table 1 summarises the pros and cons of each approach. For each measurement,

	Direct communication	Tracing	Counters/integrators
Available range of performance data	basically unlimited	wide	only summary information
Access to state info	yes	no	no
Event filtering	yes (in ULIBS)	yes (in ULIBS)	no
Monitoring overhead (NAS mg benchmark)	19.9%	5.7%	3.9%
Processing in ULIBS	high	high	low

Table 1. Comparison of three ways for gathering performance data

we use the *least intrusive* approach that still meets our goals. E.g., to carry out a system-wide analysis of the message volume sent, all the necessary support can be provided with counters with the least possible intrusion. On the other hand, when limiting the analysis e.g. to certain source functions or to the messages sent to individual destination processes only, we have to use traces (all the events are stored in the trace, together with the destination processes' IDs; the uninteresting ones are later filtered out in ULIBS). The current set of event services defined by OMIS do not efficiently support these more sophisticated performance measurements. To overcome this difficulty, new specialised event services are needed. Such services can be added to the OCM using the OMIS extension interface. The "Monitoring overhead" in Tab. 1 has been calculated for the communication-intensive NAS mg benchmark [9], relatively to the standalone PVM version, on a network of 3 Sun SparcSTATION *sun4c* class machines running Solaris 2.6.

4 PAEXT - a New OCM Extension

For the purposes of performance analysis using the counter/integrator approach, a new extension (PAEXT) to the OCM has been defined. In PAEXT two additional types of tokens, i.e., OMIS object identifiers, are defined:

`pa_c_` and `pa_cl_` for *counters*. These are plain **integer** variables that can be used for measuring values like message count and message volume.

`pa_i_` and `pa_il_` for *integrators*. These are used for storing **floating** values, but are more complicated than that. They actually store an *integral* of a function which is constant over some interval. Internally, the integrator consists of three variables: **value** — the value of the function, **time** — the beginning of the interval, and **integral** — the integral value at the beginning of the interval. The integrators can be used for measuring values like the message receive delay.

These objects can be either *local* (`pa_cl_` and `pa_il_`), i.e. there is a separate instance for each process of the application, or *global* (`pa_c_` and `pa_i_`), i.e. one instance for the whole application (for performance reasons they are implemented as local objects on each node with a global sum performed on read inside the OCM). Table 2 lists the operations defined for local objects. The same operations also exist for global ones. In the case of adding a new interval to an integrator,

Service	Meaning
<code>token pa_counter_local_create (token* objlist)</code>	create counters
<code>void pa_counter_local_delete (token* counter_list)</code>	delete counters
<code>void pa_counter_local_increment (token counter, integer increment)</code>	increment the value of the counter
<code>integer pa_counter_local_read (token* counter_list, integer clear)</code>	get values of counters
<code>token pa_integrator_local_create (token* objlist)</code>	create integrators
<code>void pa_counter_integrator_delete (token* integrator_list)</code>	delete integrators
<code>void pa_integrator_local_increment (token integrator, floating increment, floating time)</code>	increment the value of the integrator
<code>floating pa_integrator_local_read (token* integrator_list, floating time, integer clear)</code>	get values of integrators

Table 2. Local *counter* and *integrator* specific services.

time marks the end of the previous interval and the beginning of the new one, and **increment** is the value of the function in the new interval (relative to the previous value). In the read operation, if **clear** is non-0, the value is reset after being read. As an example, to measure the total delay in `pvm_recv()`, the following services can be defined:

- event services (the integrator’s value is initially 0):


```
thread_has_started_lib_call([], "pvm_recv"):
    pa_integrator_global_increment(pa_i_1, 1, $time)
thread_has_ended_lib_call([], "pvm_recv"):
    pa_integrator_global_increment(pa_i_1, -1, $time)
```

 the value of the `pa_i_1` token is increased by 1 at the start of `pvm_recv` and decreased at the end. The `time` passed each time is used to calculate the new integral, increasing it by the time spent executing the call.
- once in a while the current integral containing the time elapsed can be obtained from the `pa_i_1` integrator token, and then the integrator gets reset:


```
:pa_integrator_global_read([pa_i_1], 0, 1)
```

5 Modifications of the ULIBS Library

ULIBS is a library of high-level performance measurement and debugging functions. It defines a level of abstraction between run-time tools and the OCM. It has been developed before the OCM for use within the PARIX parallel environment, and was later adapted to the OCM. ULIBS provides the following functionality to the tools: *asynchronous* requests and callbacks, configuration and current state *info* (nodes, processes), *starting/stopping* processes, *debugger-oriented* mechanisms like symbol tables, variable info, breakpoints, single stepping. *Performance analysis* support in ULIBS comprises several functions:

- define a new measurement based on a specification passed as argument. Available kinds of measurement include *global sum*, *matrix*, *distribution*, *restriction* and *event trace*. Data to be measured includes *busy* and *sojourn time*, message passing *delays*, and *data volume*.
- start/stop the gathering of performance data,
- get performance data over the period of time passed as an argument; call an asynchronous callback when done,
- delete a measurement.

While the interface of these functions basically remained unchanged w.r.t. to PARIX version, their implementations had to be completely redesigned for the use with the OCM. As outlined above, in contrast to the highly specialized PARIX monitoring system [4], the OCM does not currently provide direct support for the more sophisticated measurement specifications, like the ability to limit a measurement to some source-level functions. These need to be emulated in ULIBS for the time being, resulting in higher overhead. More sophisticated support for performance analysis from OCM is underway.

6 Concluding Remarks

In this paper, we have presented porting two performance measurement tools – PATOP and TATOO – to the OCM environment and the enhancement of their

functionality. As a result we have got relatively universal tools which may inter-operate with other tools (e.g. debuggers). The tools may be used for the effective user-controlled search of performance bottlenecks in message passing programs. The extensions made to the OCM and ULIBS also enable to detect performance problems with further, e.g. automatic tools [10] on top of OCM.

Acknowledgements. This research was carried out within the German-Polish collaboration and it was supported in part by the AGH grant.

References

1. Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Irvin, R.B., Karavanic, K.L., Kunchithapadam, K., and Newhall, T.: The Paradyn Parallel Performance Measurement Tool, *IEEE Computer*, vol. 28, No. 11, November, 1995, pp. 37-46
2. Geist, A., et al.: PVM: Parallel Virtual Machine. A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, Massachusetts (1994)
3. MPI: A Message Passing Interface Standard. In: Int. Journal of Supercomputer Applications, **8** (1994); Message Passing Interface Forum: MPI-2: Extensions to the Message Passing Interface, July 12, (1997)
<http://www.mpi-forum.org/docs/>
4. Wismüller, R., Oberhuber, M., Krammer, J. and Hansen, O.: Interactive Debugging and Performance Analysis of Massively Parallel Applications, *Parallel Computing*, **22** (1996) 415-442
<http://wwwbode.in.tum.de/~wismuell/pub/pc95.ps.gz>
5. Borgeest, R. and Dimke, B. and Hansen, O.: A Trace Based Performance Evaluation Tool for Parallel Real Time Systems, *Parallel Computing*, **21** 4 (1995) 551-564.
6. Ludwig, T., Wismüller, R., Sunderam, V., and Bode, A.: OMIS – On-line Monitoring Interface Specification (Version 2.0). Shaker Verlag, Aachen, vol. 9, LRR-TUM Research Report Series, (1997)
<http://wwwbode.in.tum.de/~omis/OMIS/Version-2.0/version-2.0.ps.gz>
7. Bubak, M., Funika, W., Iskra, K., Maruszewski, R., and Wismüller, R.: OCM-Based Tools for Performance Monitoring of Message Passing Applications. In: Sloot, P., Bubak, M., Hoekstra, A., Hertzberger, B., (eds.): *Proc. Int. Conf. High Performance Computing and Networking*, Amsterdam, April 12-14, 1999, Lecture Notes in Computer Science **1593**, Springer, 1999, pp. 1270-1273.
8. Cunha, J., Lourenço, Vieira, J., Moscão, B., and Pereira, D.: A Framework to Support Parallel and Distributed Debugging. In Sloot, P., Bubak, M., Hertzberger, B., (eds.): *Proc. Int. Conf. High Performance Computing and Networking*, Amsterdam, April 21-23, 1998, Lecture Notes in Computer Science **1401**, Springer, 1998, pp. 708-717.
9. Bailey, D., et al.: NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, March (1994)
10. Espinosa, A., Margelef, T., and Luque, E.: Automatic Detection of PVM Program Performance Problems. In Alexandrov, V., Dongarra, J., (eds.): Recent Advances in Parallel Virtual Machine and Message Passing Interface, *Proc. 5th European PVM/MPI Users' Group Meeting*, Liverpool, UK, September 7-9, 1998, Lecture Notes in Computer Science **1497**, Springer, 1998, pp. 19-26.

Performance Modeling Based on PVM

Hermann Mierendorff and Helmut Schwamborn

Institute for Algorithms and Scientific Computing (SCAI)
GMD – German National Research Center for Information Technology
Schloss Birlinghoven, D-53754 Sankt Augustin, Germany

Abstract. For performance modeling of message passing programs using PVM, we consider a hybrid method where large numerical parts of the program are replaced by delays and the remaining control structure is translated into the input language of a simulator. For simulation, we use again a PVM-like platform. The present paper deals mainly with special aspects of the PVM layer used for simulation. As a case study, a subset of PVM was implemented on top of Modarch which is a commercial tool for simulating parallel systems. It was used here as a simulator for performance models of parallel Fortran programs. Some experiences with this implementation are discussed. This way of performance evaluation is compared with a method where the model is written in the same language as the original program and the simulator consists of a set of routines complementing the original PVM layer.

Keywords: Automatic performance modeling, parallel programming.

1 Introduction

A performance model of an application program is a coarse version of the original program showing the same timing behavior when evaluated by an appropriate tool. In the present paper, we concentrate on Fortran programs which are parallelized using PVM (cf. [1]). The user of performance models expects in general that a model is shorter regarding the number of statements and that model interpretation is faster than a run of the original program. This can be achieved only, if most of the program statements are stripped off during model generation. For performance modeling we replace, therefore, all program segments the runtime of which can automatically be estimated by delays. The remaining parts of the original program consist mainly of control statements which we translate into the control structure of the performance model. For message passing programs, the parallel program structure in particular the calls of communication routines mostly belong to the model. Therefore, a model PVM layer is required for translating the PVM constructs.

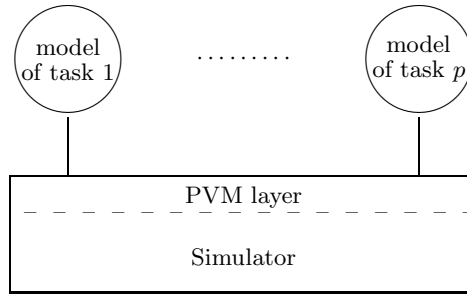


Fig. 1. Simulator of the hardware system.

There are mainly two ways for performance modeling which we call *out-language modeling* if the programming language and the modeling language (input language of the simulator) differ or *in-language modeling* if the modeling language is a subset of the programming language. Figure 1 shows a schematic view of the parallel tasks and the simulation system in the case of out-language modeling. A simulation consists of model interpretation and calls of library routines. We use Modarch as simulator which is a commercial product of Simulog.

Modarch shows some send/receive primitives but they differ from PVM. These routines do not support a message tag. The primitives tell the simulator how many data have to be transmitted, but the transmission itself does not take place. The additive part which has to be provided in the model PVM library has to handle the real data transmission for control variables existing in the model and the administration of tags.

In-language modeling means that the model is written in the same language as the original program. In this case, the complete PVM is available for handling real data transmission. Simulation of model time and in particular the virtual data transmission is, however, not supported. A simulator has to be produced which handles the related actions. Figure 2 shows a schematic view of the task system for this case. Simulation means in this case running the model as a Fortran/PVM program together with the simulator. Each simulator call requires a PVM message in the beginning. Some PVM calls require more than one message which will be discussed later.

Performance simulation of this kind is a parallel program, however, the parallelity is strongly restricted as long as only one simulator task is used.

A tool (Augur) has been developed which is able to support the user during model generation (see [2,3]). The model generation is widely automatic. The user has to define the *model parameters*, to declare *model variables*, and to define values of some *control variables*. At present, Modarch is used for back end. Important requirements for the message passing layer of this simulation are discussed in Section 2. Section 3 deals with the main features of the model PVM library implemented in the case of out-language model generation. In addition a simple example of a model is considered. The rationale of in-language modeling

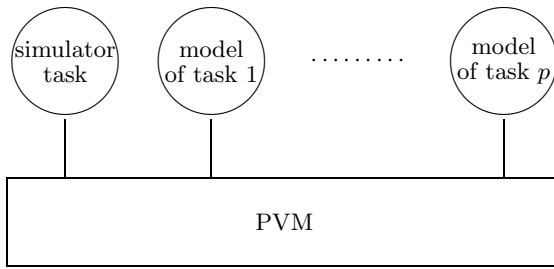


Fig. 2. Simulation of the system in the case of in-language modeling.

planned with Augur is presented in Section 4. The paper is finished by some concluding remarks.

2 Requirements for the Message Passing Layer

On top of the existing message passing layer (send/receive routines of the simulator for out-language modeling or the original PVM for in-language modeling) an interface is required which supports the missing capabilities. There are several differences between PVM and an appropriate model PVM. We are going to discuss the main differences only. Two general problems have been identified.

The first problem is the identification of the current task. A task number is required for speeding up the access to task related data like message buffers or common blocks. For out-language modeling based on Modarch, there is not even a possibility to identify the current task. Therefore, a task number has to be defined during the initialization phase of a task which is directly available within each module of the task.

The second general problem is that we have to distinguish between a *virtual* and a *real transmission* when executing a model PVM send. The virtual transmission corresponds to the amount of data having been transmitted in the original program. Only the size of this transmission is still of interest for model evaluation. The real transmission is a possibly small part of the virtual transmission concerning model variables which have to be maintained in the model program. This transmission has of course to take place, i.e. the content of the message has to be moved from the sender to the receiver.

3 A Case Study for Out-Language Modeling

3.1 Automatic Performance Modeling Using Augur and Modarch

The simulation of performance models of Fortran programs by Modarch is an example for out-language modeling. The simulator behind Modarch is QNAP2 the input language of which is similar to Pascal. Only a subset of the general features of Pascal is available with QNAP2. The Modarch model of a given Fortran program can automatically be generated using Augur. Augur translates

the control structure of a program into a performance model and replaces large computational blocks by appropriate delays.

The model PVM layer of Augur is a simple subset of PVM. All point to point and multi-cast communications, asking for task-ids, spawning and killing of tasks are included. Only the master-slave model of parallel work is supported, i.e. a master task is started at the beginning and all other tasks have to be started by spawning. The tasks need not be identical. This depends on the code which is 'loaded' into the simulated processors at the beginning using the Modarch editor. Instead of starting a PVM daemon in the beginning, the simulator starts the master.

3.2 Fast Identification of the Current Task

The scope of variables of QNAP2 programs is global to the whole program or local to a procedure and the task the procedure is currently running for. Data which should be visible to all procedures or all tasks (common blocks or send/receive buffers) can only be realized in a global array. For identification of the current subset of that array, an index has to be maintained which is a local variable of the current task. A unique task number is defined in the very beginning of a task by the call of a routine which enumerates the PVM tasks. The call delivers the task number as value of a local variable `pvmqmytid` which is made available by using an additional parameter at the beginning of the parameter list of all subsequent procedure calls.

3.3 Packing of Virtual and Real Data

All pack or unpack routines show an additional mode parameter at the end of their parameter list specifying whether the data have to be moved or have to be counted only. The majority of data should of course be counted only for keeping models small. In particular all big floating point arrays used for numerical computations may be of this type. Control data like arrays of task-ids are typical candidates for being transmitted among the tasks of the model.

3.4 Special Problems with Pointers

The Fortran version of PVM routines allows to specify the beginning of data by a pointer which is actually defined by an array name, a scalar, or an array element. Even the type of this pointer can be any data type of Fortran. This is not allowed for QNAP2 programs. Depending on the data type, `pvmfpack` has been replaced by `pvmf xy pack` where xy stands for `is` (integer scalar), `ia` (integer array), `rs` (real scalar), `ra` (real array), etc. In the case of packing more than one data item, the array name has to be inserted as parameter followed by a list which contains an appropriate descriptor. This list has the form

(rank, begin₁, offset₁, end₁, begin₂, offset₂, end₂,..., terminator)

$begin_k$ and end_k describe the declaration of the corresponding array for dimension k while $rank$ is the rank of the array. The descriptor allows adaptation of different memory mapping of data in Fortran and QNAP2 (Pascal) in addition. Unpack routines are handled correspondingly. The same description is also used for the corresponding parameter of `pvmfspawn`.

3.5 PVM Parts of a Jacobi Relaxation for Example

We consider the model of a program which solves the 2D Laplace equation by Jacobi relaxation on a 1D process array for example (see [4]). Only some model segments are shown which are of interest here. The model is generated by Augur. For more details, we refer to [3].

The following code segment is the spawning part of initialization for a parallel system of p processes.

```
pvmfspawn(pvmqmytid,
  "jacobi",0,"*",p-1,proc(pvmqmytid).tids,(1,1,2,p,0),numt);
& the error handling (numt<p-1) is left out
IF (1*(p-(2)) >= 0) THEN
FOR actproc := 2 STEP 1 UNTIL p DO
BEGIN
  pvmfinitsend (pvmqmytid,0,info);
  pvmfispack (pvmqmytid,3,actproc,(0,0),1,1,info,1);
  pvmfiapack (pvmqmytid,
    3,proc(pvmqmytid).tids,(1,1,1,p,0),p,1,info,1);
  pvmfsend (pvmqmytid,proc(pvmqmytid).tids(actproc),1,info);
  COMPUTE( 0.0 );
END;
```

`tids` is an integer array of rank 1. The task-ids of the slave tasks are stored with indices 2 to p . In the Fortran program, the pointer was simply `tids(2)` and the descriptor was not required. The loop sends the logical task number and the `tids` array to the slaves. The array `tids` itself is here part of a global array structure `proc` which represents a common block of the original program. As the last parameter is 1, the pack routines work in the real data transmitting mode.

The local communication for exchanging boundary data between computational steps of the iterative solver is represented by the following code segment. `proc(pvmqmytid).myproc` contains the index of a task in `tids`. Knowing this, the tasks are able to address their left and right neighbors for communication (see example below). The routine sends virtually n data items left and right and receives them left and right. In contrast to the above code example, no data are transmitted by the simulator because the last argument of the pack and unpack routines is 0 but the size of transmission is considered for the simulation time. Though no data are transferred, a pointer `u` and an empty descriptor `(0,0)` have to be present in pack and unpack routines for formal reasons.

```

IF (proc(pvmqmytid).myproc > 1) THEN
BEGIN
  pvmfinitsend (pvmqmytid,0,info);
  pvmfrapack (pvmqmytid,4,u,(0,0),n,1,info,0);
  pvmfsend (pvmqmytid,
    proc(pvmqmytid).tids(proc(pvmqmytid).myproc-1),2,info);
END;
.... & analogue for the right side using myproc+1 for address
IF (proc(pvmqmytid).myproc > 1) THEN
BEGIN
  pvmfrecv (pvmqmytid,
    proc(pvmqmytid).tids(proc(pvmqmytid).myproc-1),2,info);
  pvmfraunpack (pvmqmytid,4,v,(0,0),n,1,info,0);
END;
.... & analogue for the right side using myproc+1 for address

```

3.6 Comparison of Measurements and Model Evaluation

The comparison of measurements and model evaluation gives an impression which accuracy can be reached. We investigate the parallel 2D Jacobi relaxation on $n \times n = 256 \times 256$ grid points on a system of SUN Ultra 5 stations connected via fast Ethernet. 100 iteration steps on $p = 1, 2, 4$ processors were considered. We achieved 7.58 Mflop/s for the processors, a start-up time of 1.60 ms for a pair of send and receive calls in a pingpong benchmark, and a maximum sustained bandwidth of 3.78 MB/s for the message transfer.

We counted only one class of operations containing floating point operations of any type. This approach corresponds to the traditional way of using a Mflop/s rate to characterize the performance of a processor.

Figure 3 shows that our performance modeling works well by comparing model evaluation and measurements for the runtime. We measured all cases 5 times because the timing values change from one run to the next one. The run-times of the original program on the SUN Ultra 5 were about 6.5, 4.2, and 2.9 ms for 1, 2, or 4 processors respectively. The simulation time on a PC (Pentium II-MMX 266 MHz) took 14, 64, and 240 s. Though we have observed a better ratio with large compute bound programs, the current simulation speed is definitely not acceptable.

4 In-Language Modeling on Top of PVM

4.1 Rationale of In-Language Modeling

A very natural way of implementing the model PVM layer is encoding in Fortran itself. Then the whole PVM layer is available for execution of the real data transfer. However a simulator is required which administrates the model time and the utilization of all simulated hardware units. The model PVM layer can

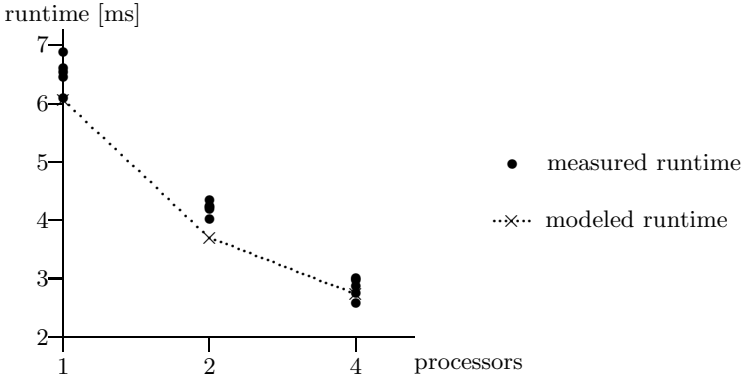


Fig. 3. Model evaluation and measurements for Jacobi relaxation.

be implemented on top of standard PVM for handling the virtual data transfer and as an interface for the simulator.

We decided to run the simulator kernel as an additional task which is spawned by the first user task (cf. figure 2). In this way, in-language modeling is a parallel simulation method from the beginning. As in the case of out-language modeling, we anticipate that each task starts by a special model PVM call which introduces the calling task to the simulator and which starts the simulator task in the very first call. Each routine which addresses the simulator has to send a PVM message to the simulator. In most cases, these messages inform the simulator on the duration of the computational work which has currently to be considered for the sending task. Some of the PVM routines (mainly send/receive routines) have to wait for the simulator's answer if they need some directives before the work can be continued.

In-language modeling is expected to run much faster than simulation with Modarch because the Fortran compiler translates the model which leads to an optimized simulation executable.

4.2 The Modeling Coherence Problem

In-language modeling brings up a new modeling coherence problem which is not existing before. We consider two messages which are send to the same target task. Let t_i , t'_i , and t''_i ($i = 1, 2$) be the clock values at which the messages arrive in the real world, the simulator or the host system respectively. t'_i is determined by the simulator and t''_i by the host system of the simulation.

The real world and the simulation are considered coherent iff $t_1 < t_2 \iff t'_1 < t'_2$ holds. This relation is satisfied if the simulator is properly designed and if the considered hardware architecture including communication network and routing strategy is specified correctly. As long as all messages are uniquely identified by the task-id of the sender and by a tag we do not have any further coherence problem. In this case, it is not of importance when the real parts of the

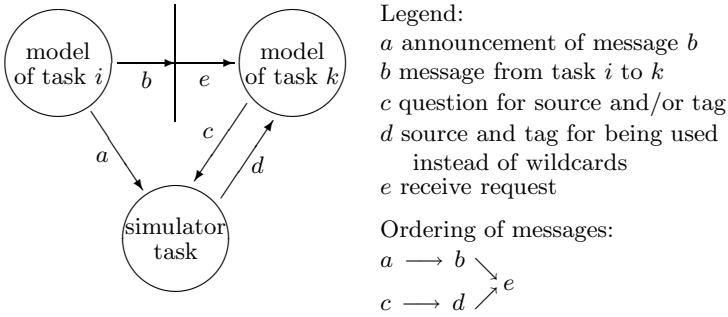


Fig. 4. Handling of send/receive in the case of wildcards.

messages become available via the PVM layer. However, if wildcards are used for sender and/or tag, the *modeling coherence condition*

$$t'_1 < t'_2 \iff t''_1 < t''_2$$

might be violated because the simulated architecture and the architecture of the host system may differ considerably. In order to guarantee modeling coherence, we follow the scheme of Figure 4 in send/receive routines of model PVM. If called with wildcards, the **receive** of model PVM asks the simulator first for source and tag and uses these instead of wildcards with the PVM routines.

5 Concluding Remarks

Two ways of modeling the performance of parallel Fortran programs using a PVM platform and an appropriate simulator have been discussed. For out-language modeling a tool set has successfully been implemented which supports model generation and evaluation. Current work is related to the implementation of a simulator for in-language performance modeling which is expected to run faster than simulation by Modarch. Future work will include extension to MPI.

References

1. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing*, (MIT Press, Cambridge, MA, 1994).
2. H. Mierendorff and H. Schwamborn, Generation of Performance Models, in D'Hollander, Joubert, Peters, Trottenberg, *Parallel Computing: Fundamentals, Applications and New Directions* (Elsevier Science, Amsterdam, 1998) 693–696.
3. H. Mierendorff and H. Schwamborn, *Automatic Model Generation for Performance Estimation of Parallel Programs*, (accepted for Parallel Computing).
4. E. van de Velde, *Concurrent Scientific Computing* (Springer-Verlag, New York, 1992).

Efficient Replay of PVM Programs^{*}

Marcin Neyman, Michał Bukowski, and Piotr Kuzora

Technical University of Gdańsk
Narutowicza 11/12, 80-952 Gdańsk, Poland
marcinn@pg.gda.pl

Abstract. The paper presents a definition of replay of a distributed application as a function of three parameters: depth, width, and length. It addresses the problem of nondeterminism in distributed system and proposes an efficient approach to trace a PVM application behaviour in order to eliminate races in repetited execution. Detecting races in distributed computations requires implementation of a strongly consistent system of vector clocks. Therefore a system of vector clocks was adapted for a dynamic application model. Finally it presents the architecture of a tool supporting replay of PVM applications.

1 Introduction

Recovery techniques are widely used in solving problems of distributed systems. The most popular use of replay methods is providing repeatability in testing and debugging of distributed programs[1,2,5,10]. There are some implementations of replay tools for PVM[9,7] based on various assumptions. A big effort has been put on achieving efficiency of tracing and replaying in distributed systems[11,3].

This work is an approach to implement an efficient replaying tool for PVM application that traces only necessary data. The tool is also capable of supporting modification of the repetited execution in order to test selected parts of an application or correct erroneous computation caused by wrong decisions.

2 Model of a Distributed System

The PVM[4] model assumes a finite set of independent applications are simultaneously working in the system. The system consists of a finite set of autonomous hosts connected by the means of a communication network. We assume no shared memory nor global clock is used. Each application consists of a group of cooperating processes. The number of processes in an application may change dynamically (the processes may be created and terminated at any point during the application life time).

The processes cooperate with each other within an application by sending and receiving messages. Interprocess communication is limited to traditional message

^{*} This work was sponsored by INCO-Esprit KIT Project no. 997100 (Parallel Processing Tools. Integration and Results Dissemination).

passing. No message box communication is allowed. Processes communicate via unidirectional logical channels. The channel identifier is a pair $chid = \langle sid, rid \rangle$, where sid is sender identifier (input to the channel) and rid is receiver identifier (output from the channel). Messages sent through the channels are labelled with integer tags that can be used to filter them when they are received. According to properties of PVM communication channels preserve order of messages and guarantee that messages will eventually be delivered.

We define the execution of a distributed application as a pair $P = \langle E, \overset{HB}{\rightarrow} \rangle$, where E is a finite set of events and $\overset{HB}{\rightarrow}$ is “happened before” relation defined over E [2].

The “happened before” relation[6] is defined as a transitive closure of the union of another two relations:

$$\overset{HB}{\rightarrow} = (\overset{EO}{\rightarrow} + \overset{M}{\rightarrow})^+$$

Where: $\overset{EO}{\rightarrow}$ — events order in one process;

$\overset{M}{\rightarrow}$ — message delivery order.

An event is the smallest action that can change a state of one process and at most one communication channel incident to this process. Events may be divided into three classes:

- internal events (do not change the state of any communication channel),
- inter-process communication events,
- I/O events (responsible for the communication with the environment).

3 Nondeterminism

A distributed application is surrounded by its environment that influences it in an unpredictable way. Influence of the environment is a function of communication network load, activity of users, hardware behaviour, and other factors. This unpredictable influence causes possible nondeterministic behaviour of the application. For this reason subsequent executions of the same application with the same input may differ although producing the same output[11,9]. On the other hand it is possible to obtain different outputs. It may be caused by an error in implementation of interprocess communication, but sometimes it is a desired effect.

Two sources of nondeterministic execution of a distributed application may be distinguished. The first one is the data input from the environment (i.e. console, files, system clock, random generator, etc.) that vary from execution to execution. The other source are communication races. In the traditional message-passing model which is assumed in this work (PVM without message boxes) this is limited to receive races[11]. Because of nondeterministic influence of the environment, potential behaviour of an application ($P' = \langle E', \overset{HB'}{\rightarrow} \rangle$) may differ from the actual one. The receive race is defined as a situation when one event is capable of receiving either of two messages sent by the separate processes.

Figure 1 shows an example of receive race between messages $a \xrightarrow{M} r$ and $b \xrightarrow{M} q$. There is a potential behaviour (figure 1(b)) where r receives the second message opposite to the actual behaviour (figure 1(a)).

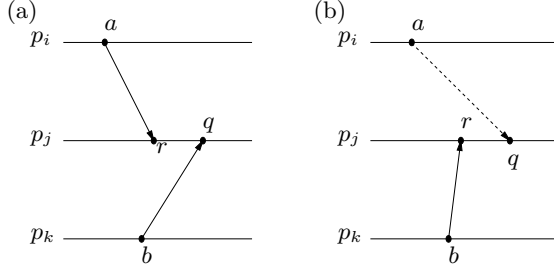


Fig. 1. Example of receive race

4 Parameters and Modes of Replay

Replay is a repetited execution of an application or its part. It may be defined as a function of three parameters[10]: depth, width, length.

Depth specifies the abstraction level of replay. The following three depth levels may be distinguished: assignment statement level, inter-process communication level, process creation level.

Width determines the range in the process space, i.e. how many processes of application are involved in the replay. Length specifies the range in the time space and is defined separately for each of the depth levels. It shows the distance from the application state to which rollback is addressed to the state that finishes replay. Table 1 shows definitions of this parameter.

Table 1. Definitions of replay length according to replay depth

Depth level	Length definition
process creation level	number of processes created
interprocess communication level	number of messages sent
assignment statement level	vector of numbers of actions performed by each process

According to the replay the following four modes of execution of an application may be described:

- primary execution,
- traced primary execution,
- real re-execution,
- simulated re-execution.

Primary execution is the standard mode in which program is run. It executes pure application code without any procedures that support replay. To log the application behaviour, tracing procedures must be added to the application. This mode is called traced primary execution.

Real re-execution uses support mechanisms that provide control over application using data logged during primary execution. The mechanisms may be implemented as procedures compiled into the application code or as external elements that control the environment influence. In the perfect case we should be able to replay pure primary execution. Unfortunately it is impossible to achieve in the real life. What we can actually replay is traced execution which is in fact an approximation of the real application behaviour.

Simulated re-execution is performed outside the real environment. It is based on interpretation of data logged during primary execution. Usage of this mode is limited because of its passive character and the lack of support for any modifications of recovered computations. On the other hand this method guarantees determinism despite of recovery depth level, what is not always possible for the real re-execution. It allows to adjust the detail level of tracing according to the demanded precision. This mode was utilised by STEPS[5] for visualisation and testing of parallel programs.

In this work we will focus mainly on the real re-execution mode.

5 Efficient Tracing and Race Detection

To replay a distributed application or its part, the following three conditions must be satisfied:

- input from the environment is recovered,
- order of message delivery is restored,
- communication with processes that are not replayed is recovered.

The first condition may be accomplished by tracing all input events and logging the data they read. The third condition requires recovering messages sent by other processes from the log recorded during primary execution or replaying suitable intervals of execution of their senders.

To avoid races in repetited execution it is necessary to trace the inter-process communication. However tracing receive events is sufficient to detect races. Moreover, not all receive events are involved in races. We can focus only on those that accept messages from many processes. The events that receive messages from a single process but accept many message tags are not candidates to race as communication channels are assumed to preserve the order of the messages. Tracing all receives that accept messages from many processes would be still inefficient because not all of them are involved in races.

An elegant solution for race detection was proposed by Netzer and Miller[11]. It is capable to detect receive races on-the-fly. Figure 2 illustrates the method. The algorithm checks whether the message $b \xrightarrow{M} q$ could have been received by different (earlier) receive event of the same process. To achieve this a previous event (r) that receives from the same logical channel is located (in PVM this means receive operation with sender $tid = -1$ and the same message tag or message $tag = -1$ (ANY flag)). A race between messages sent by a and b exists when actions r and b are concurrent ($r \parallel b$). The ordering of actions is determined by comparing their vector timestamps[12]. The method described in the mentioned work is based on tracing of the second message involved in the race. We present a solution that traces the first message. We claim this approach easier to implement in the PVM environment while being not less efficient. Implementation details will be described in the next section.

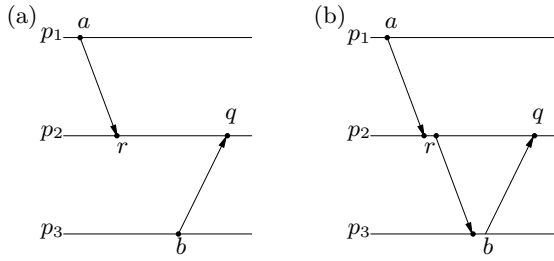


Fig. 2. Race detection: (a) race exists ($a \parallel b$), (b) race does not exist ($a \xrightarrow{HB} b$)

6 The Replaying Tool Architecture

The method described in the previous section was implemented. The tool was written in PVM and as such is aimed at the PVM programs. General guidelines for this implementation were not to change any elements of PVM nor the application being replayed.

The tool consists of four parts:

- Recorder** — centralised process logging the information about the primary execution; receives registrations from newly created processes and data on racing messages;
- Replayer** — centralised process coordinating replay procedure; receives registration from newly created processes and sends to the processes synchronisation data used to avoid races;
- ReLib** — library linked to replayed application code; contains wrappers to the `main()` function and some of the PVM functions (`spawn` and `receive`).
- Log** — centralised file containing traced information on process creation and racing messages.

Figure 3 presents the tool architecture.

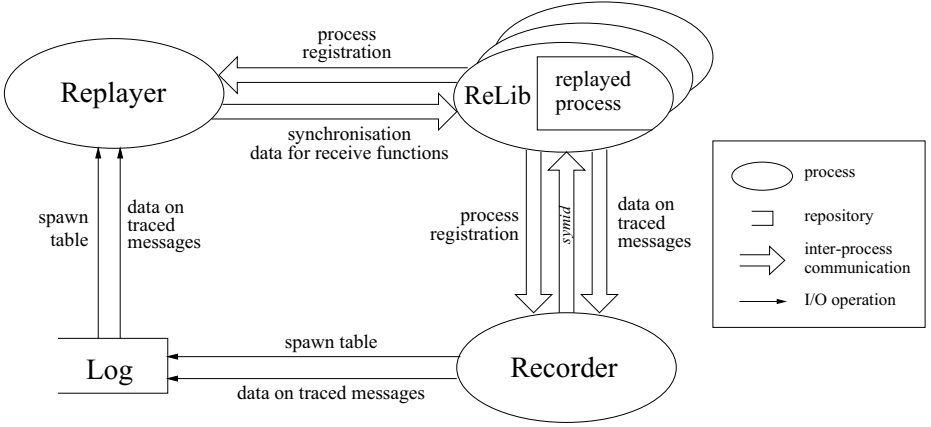


Fig. 3. Architecture of the replaying tool

6.1 Process Identification

Process identifiers in PVM (*tid*) vary from execution to execution. Logging information on an application behaviour needs to be based on symbolic identifiers (*symid*) that do not change between executions. To achieve that we need to trace the creation of processes. To be able to assign a *symid* to a *tid* during replay the following information must be logged during the primary execution:

- *symid* of the parent process,
- line number for the appropriate spawn instruction,
- *symid* of the spawned process.

Two approaches for tracing the process creation may be proposed:

- registration is performed by each of the newly spawned processes;
- the processes executing *spawn* register their children.

In the first approach the newly created task must receive information about line number of the instruction that spawned it. This requires sending an additional message from the parent to its child, what increases the probe effect and thus should be avoided. That is why we decided to use the other approach. Each process after executing *spawn* sends a *register* message to the *Recorder*. The message contains:

- the process *tid*,
- the line number of the spawn instruction,
- a list of *tids* of the spawned processes.

During replay the *Replayer* reads the *Log* file, builds the process creation tree, and assigns *symids* to *tids* of registered processes. Every process during its initialisation receives a message containing its *symid*.

The approach we decided to use requires special treatment for processes created manually (not spawned by other process). Those tasks are not registered by their parent. Each process during its initial phase checks its parent id. If it receives a negative response it has to register itself by sending a message to the *Recorder*. In this case the data on parent (*tid*, line number of spawn) are set to zero, only the spawned task *tid* is meaningful.

6.2 Tracing the Racing Messages

During primary execution a race detection procedure is performed. It requires a method to determine casual order of events. The approach described in [11] is based on vector timestamps. Traditional system of vector clocks[12] assumes a static number of processes constituting an application. Therefore we had to adapt the system of vector clocks to the dynamic application model. It is based on the following statements:

- every process during its initial phase is assigned a subsequent non-negative integer (*symid*);
- the local vector clock of the process p_i is initialised as a vector vt_i of size i ;
- on receive of a message with a vector timestamp vt_m piggybacked, the size of local clock vt_i is set to $size(vt_i) = \max(size(vt_i), size(vt_m))$.

The processes maintain their vector clocks updated with every inter-process communication event and local counters. Local counter for the process p_i ($sid = i$) is equal to the i^{th} slot of its vector clock. All messages piggyback a timestamps equal to the local vector clocks of their senders. We will refer to the figure 1. According to the method described in section 5, after executing a receive event (q) a previous receive (of the same process) with $tid = -1$ and the same tag or $tag = -1$ is located (r). To check whether the previous receive (r) we have located happens before the send event (b) the local counter of this receive is compared to the piggybacked timestamp. If the local counter of r is greater then the j^{th} slot of the timestamp r does not happen before b so race exists. The message $a \xrightarrow{M} r$ will be traced. The message sent to *Recorder* actually contains *tid* of the sender (p_i) and the message *tag* as well as receive *tid* (p_j) and the serial number of the receive function.

6.3 Replay Procedure

During repetited execution the *Replayer* reads the *Log* file and sends to each replayed process a message consisting of a table of pairs: (sender *tid*, message tag). Sometimes is not possible to send the whole table because some processes are not yet created and their *tids* are not already known. In this case the table must be sent in parts. The table is received using message handler mechanism.

Parameters of original *pvm.*recv()* functions are replaced with the ones read from the table. This action ensures that the same message will be received as it has been during the primary execution.

7 Final Remarks and Future Work

The tool described above is a prototype used to perform experiments on replaying distributed computations with various values of recovery width and depth. At the moment it has some limitations that are subject for the future work. The most important improvement will be ability for incremental replay. This requires implementation of checkpointing procedures. The next step will be expanding the tracing procedure for detecting races in message box communication.

References

1. Dione, C., Feeley, M., Desbiens, J.: A Taxonomy of Distributed Debuggers Based on Execution Replay. Proc. of the International Conference on Parallel and Distributed Techniques and Applications, Sunnyvale, California (1996)
2. Damodaran-Kamal, S.K., Francioni, J.M.: Testing Races in Parallel Programs with an OtOt Strategy. Proc. of the 1994 International Symposium on Software Testing and Analysis (ISSTA), ACM Sigsoft, ACM Press, New York (1994) 216–227
3. Fagot, A., de Kergommeaux, J.C.: Systematic Assessment of the Overhead of Tracing Parallel Programs. Proc. of PDP'96, IEEE Computer Society, (1996) 179–186
4. Geist, G.A., Beguelin, A., Dongarra, J.J., Jiang, W., Manchek, R., Sunderam, V.S.: PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing. MIT Press, Cambridge, MA, (1994)
5. Krawczyk, H., Wiszniewski, B., Kuzora, P., Neyman, M., Proficz, J.: Integrated Static and Dynamic Analysis of PVM Programs with STEPS. Computers and Artificial Intelligence, 17(5) (1998) 441–453
6. Lamport, L.: Time, clocks and the ordering of events in a distributed system. Communications of ACM, 21(7) (1978) 558–565
7. Lourenço, J., Cunha, J.C.: Replying Distributed Applications with RPVM. Proc. of DAPSYS'98, (1998)
8. Lourenço, J., Cunha, J.C., Krawczyk, H., Kuzora, P., Neyman, M., Wiszniewski, B.: An integrated testing and debugging environment for parallel and distributed programs. Proc. of the 23rd Euromicro Conference (EUROMICRO'97), IEEE Computer Society Press, Budapest, Hungary, (1997) 291–298
9. Mackey, M.: Program Replay in PVM. Technical Report, Hewlett Packard, Concurrent Computing Department, Hewlett Packard Laboratories, (1993)
10. Neyman, M.: Non-deterministic Recovery of Computations in Testing of Distributed Systems. Proc. of Ninth European Workshop on Dependable Computing, (1998) 114–117
11. Netzer, R.B., Miller, B.P.: Optimal Tracing and Replay for debugging message-passing parallel programs. The Journal of Supercomputing, 8(4) (1995) 371–388
12. Raynal, M., Singhal, M.: Logical Time: Capturing Causality in Distributed Systems. IEEE Computer, 1 (1996) 49–56

Relating the Execution Behaviour with the Structure of the Application

A. Espinosa, F. Parcerisa, T. Margalef, and E. Luque

Computer Science Department
Universitat Autònoma de Barcelona.
08193 Bellaterra, Barcelona, SPAIN.

Phone: +34 93 581 19 90, Fax: +34 93 581 24 78

e-mail: {a.espinosa,f.parcerisa,t.margalef,e.luque}@cc.uab.es

Abstract. Traditional parallel programming forces the programmer to understand the enormous amount of performance information obtained from the execution of a program. In this paper, we show how the use of KappaPi automatic analysis tool helps the programmers of applications to avoid this difficult task. In the last stage of the analysis we discuss the possibilities of establishing relationships between the performance information found and the programming structure of the application.

1 Introduction

The main reason of using parallel systems is to get good performance when executing applications[1]. Users decide to use a parallel system for executing applications in order to get the best possible performance, reducing the execution time of the application as much as possible.

Unfortunately, to assure the best use of a machine, performance hungry users must analyse the behaviour of the application. Moreover, to solve reasonable questions like how is the program performing or why is the application performing poorly the programmer must face the hard task of becoming an expert in the performance analysis.

Classically, users of parallel systems execute their applications together with some monitoring facilities provided in the system to generate a trace file[2,3,4,5]. This file records all information generated during the execution. Then, users must use some visualisation tool to analyse the behaviour of the program. The analysis views typically let the programmer focus on a determinate region of the execution and visualise some detailed information about it. Some important informations displayed include the messages sent between the processes and the execution state of the processors. As an important feature, there is the possibility to zoom in and out of execution intervals.

This work was supported by the CYCYT under contract number: TIC98-0433.

J. Dongarra et al. (Eds.): PVM/MPI'99, LNCS 1697, pp. 91-98, 1999.

© Springer-Verlag Berlin Heidelberg 1999

These views give execution information that should be relevant to experienced programmers. They are meaningful only if the programmer knows how to relate this detailed information with the actual program. This is, by no means, an easy task and requires a very good knowledge of the execution consequences of using every message-passing primitive. The low detail and the high amount of available information make difficult the possibility to abstract the necessary information to understand what are the consequences of the programming decisions.

Therefore, in many cases, parallel programmers need to spend a lot of time and effort to reach a desired performance index.

To avoid all these big efforts and save user time, it would be necessary to provide a tool that makes the performance analysis, detects the execution bottlenecks and relates those problems with the high level structures of the application [6,7,8,9]. Although this is in a more advanced stage on the Fortran programs [10] due to the more determined characteristics of the data involved.

Experience with the visualisation of execution [11] leads to the creation of some performance problem categories that include typical problems like load imbalance, synchronisation blockings, ...

These categories allow the automatic recognition of each class of problems and the possibility to recommend the user some alternatives of implementation to avoid the problem. However, the analysis is still based on the processing of rather low level details like the events of a typical trace file. This dependency forces the user to understand the impact of the parallel language primitives on the machine when executing the program.

It would be very interesting for users to receive some information about the behaviour of the program based in their own program structures. In this work, we want to show some existing links between the low-level performance problems and some typical application implementations. These links can be used in order to generate more significant program performance information.

In the following sections, we describe the steps that are needed to analyse the performance of a PVM [12] parallel program with the use of KappaPi tool[13]. In section 2, we introduce the efficiency analysis done by the tool. Section 3 presents the advantages of including some references to source code from the execution information. In section 4, we discuss the possibility of establishing relationships between the execution trace data and the program design. Finally, in section 5 we present our conclusions.

2 Efficiency Analysis

KappaPi tool defines the performance quality in terms of the processor efficiency along the execution. That means, if a processor is idle during a certain execution

interval, this would be analysed as a performance problem. Idle intervals can contain waiting-for-a message blocks, synchronisation waits or other problems.

Once the performance quality has been determined, KappaPi tool starts the interpretation of the trace file in terms of its efficiency. That means the tool will look for those execution intervals where the efficiency is lowest. These intervals will be recorded depending on their impact on the global performance. That is, the tool will always record those problems that affect longest and to the most number of processors.

3 - Complementing Trace File Information with Source Code References

At this point of the analysis we already know the most important problems of the execution, but only in terms of inefficiency. Now is the time to magnify the details of the execution interval. With the help of the source code references of the trace file events, the tool can rebuild the execution scenario of the application. The objective is to establish all possible links between the behaviour of the program (normally expressed in terms of primitive calls) and the programming structures of the code.

After having determined the most important problems, KappaPi tool classifies the execution situations found under some previously defined categories (expressed in table 1). These categories discriminate mainly between communication related and program structure problems.

Table 1 : Brief description of the low-level performance problems detected when analysing the trace file

Problem	Description
Sequentiality	Some processes are serialising the execution of the application.
Load imbalance	There is a significant difference in the processes assigned to different processors
Barrier problems	The use of barrier primitives blocks the execution
Slow communication	Individual (or burst) communications slow the execution of the application.
Bottleneck communication dependencies	The communication behaviour of the application provokes different blocking situations regarding an ill-behaviour data dependence.

KappaPi tool is mainly based in the identification of one of these categories for each performance problem under analysis. When a performance problem is classified, source code details like variable names and data structures in use are identified. With the help of these informations, the KappaPi tool, depending on the problem, is able to

test some source code details of the application related with the execution. For example, if the problem found involves some communications, the tool tests the data dependencies between the communicating processes.

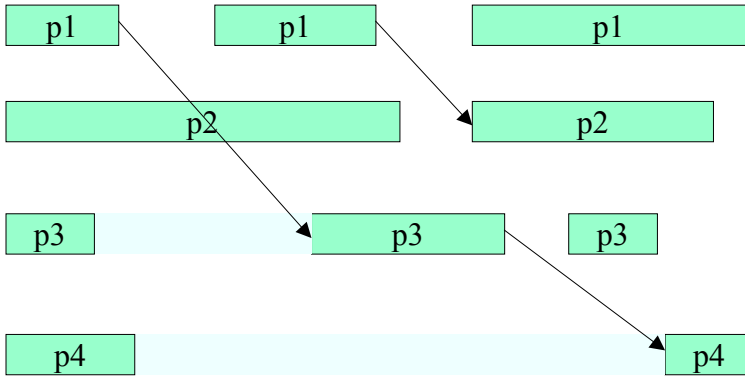


Fig. 1. Classifying a communication dependency

In the example of figure 1, the problem found is a low level communication (the inefficiency is found when process p3 and p4 are waiting for a message). As p4 is waiting for p3 while p3 is also waiting, the problem is classified as a possible bottleneck communication dependency.

The next is to look for more information in the trace file to complete the description of the problem. In figure 2, we express the source code of p3. In that code we may take notice of the reception of the variable "calc", which is sent to p4. Then, we have discovered a dependency of the program that can help to understand the execution problem found previously.

```

1 pvm_recv(-1,-1);
2 pvm_upkfl(&calc,1,1);
3 calc1=min(calc,0);
4 for(i=0;i<sons;i++){
5     pvm_initSend(PvmDataDefault);
6     pvm_pkfl(&calc1,1,1);
7     pvm_send(tid_son[i],1);
8 }

```

Fig. 2. Source code of process p3

4 One Step Beyond, Is It Possible to Read the Programmer's Mind?

Performance analysis, although not visualised, is still based on the processing of the trace file events. The next step is to recognise the actual execution of some typical programming structures with the help of the previous layers of already obtained information (selected trace events and related source code information).

Parallel programmers, when designing applications, decide which is the best way to distribute code and data in order to implement the desired algorithm. The following step is to implement those design decisions with an actual parallel programming language and added libraries. Programmers must decide which programming environment is better for the application considering some issues like familiarity with the language or platform availability.

The decision of using a specific programming environment will definitely influence the actual performance of the application. The use of a certain programming language (plus libraries) facilitates some programming structures and complicates the creation of others.

For instance, the use of certain message passing defines one determinate subset of parallel applications. Those applications have a "typical" way of using the language primitives, mainly because there are some easy ways of building up a parallel application with those primitives.

If we can build a description of such a typical behaviour of an application under some known programming details (like language and message-passing library used), this description can be used, not only to identify its components but to look at the behaviour of the application using this structure. Furthermore, this description can be used to build some suggestions about improving the performance of the application, provided that the functionality of this part of the application is basically known.

4.1 A Practical Example: Master/Slave Programs in PVM

By that, rather indeterminate, description we want to define the master/slave application designs that fit in the following roles: there is one kind of process (the masters) that generate some data items and, immediately send them to another group (the slaves) to be processed. As soon as the data is sent, the masters start producing data to create a new message.

On the other hand, the slaves need the data generated in order to finish some calculation or simply to resume their execution. They will wait blocked when they require new data from the "masters".

The non-blocking send, the blocking receive and the use of the “spawn” primitive (dynamic creation of processes) define a specific way of distributing any calculation along the processes of a virtual machine of PVM environments.

Firstly, PVM provides us with the “spawn” function to create new processes (returning the new process identification). Provided that processes cannot send a message to those with unknown identification, each master process will spawn the slaves processes that is going to work with.

The straightforward way to program a master/slave application in PVM is to use “private” slaves. I. e., each master creates, whether statically or dynamically, the slaves it needs to carry out the computation. Once they are created, the easiest way to communicate with the slaves is asynchronously, that is, “send and forget”. Hence, the slave processes wait and process the data they receive, while the masters send and immediately generate new data messages.

The detection of an actual master/slave is a difficult matter. First of all, there is not a unique way of writing an application with the above characteristics. In addition to this, the parallel programming libraries usually provide ambiguous primitives that allow the creation of similar behaviours with very different implementation details.

Secondly, any analysis of the application must consider the scope of the problem found into the global execution point of view. The application may show a master/slave problem at some execution time interval, but the programmers may not be aware of that behaviour when the application was designed.

What we know from the typical construction lets us look for the following behaviours:

As both master and slave processes send and receive messages from each other, we may understand that, if any two processes send message to each other, we will deduce that those processes are related. This relationship is the start point for the analysis of any master/slave problem. Once all collaborating processes have been identified, it is time to analyse where is the problem and how is this problem created.

What we know until now is that we have a performance problem and that we have found a relationship between some processes. Now we must look for the blocking processes that are creating the performance problem. We will look for those processes, included into the send/rcv relationships, that are blocked waiting for a message. If a process belongs to the relationship and it is blocked waiting for a message, then it is supposed to collaborate as a slave.

After this, all slave “candidates” are analysed in order to find whether they fit in the slave role. That is, if the process is receiving the same data item, performs some calculation and waits for another. If this is found true, the process will be classified as a slave and the sender of the messages, as a master.

Finally, after all masters and slaves have been identified, the resulting construction is going to be analysed as a whole, in terms of its inefficiency.

We now have a list of blocked processes. If we can explain the reason of their delay, we may find how to solve it. Then, we can propose a way to execute this master/slave better, without changing its initial behaviour.

If the messages sent are big (the waiting for arrival time is not meaningless compared to the complete waiting time. This value depends very much on the interconnection network situation) the user must be warned to reduce the length of the messages.

If the message generation rate is smaller than the message consumption rate (there are "bored" slaves, see figure 3), the number of slave processes must be reduced. I. e., the user must change the design to use less slaves. This decrease must be as big as the rate difference.

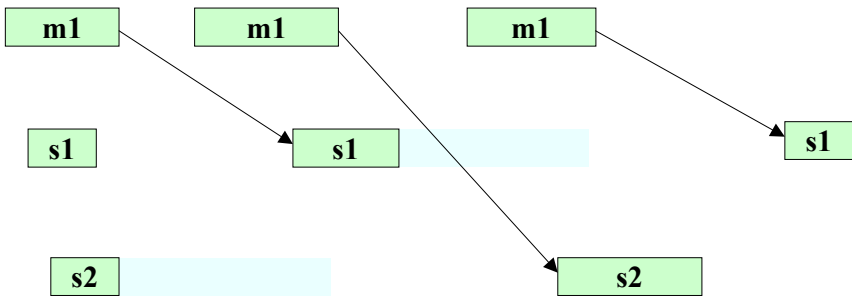


Fig. 3. Trace level snapshot of a "bored slaves" problem. Both process s1 and s2 wait blocked for the reception of a message from m1.

If the slaves keep processing data after the master processes have finished, the message generation rate is bigger than the message consumption rate (messages queues at the receivers are full), the number of slaves must be increased.

<u>master code:</u>	<u>slave code:</u>
<pre> num=spawn("slave",0,0,"", NUMSLAVES,&sontids[0]); for(i=0;i<NUMMESS;i++){ <data generation> pvm_pkfl(&data,1,1); pvm_send(sontids[i],1); i=(i+1)%num; }</pre>	<pre> my_tid =pvm_mytid(); while(!end_recv){ pvm_rcv(-1, -1); pvm_upkfl(&data,1,1); while(data>0){ if(var==-1) fin =1; <data consumption> } }</pre>

Fig. 4. Source level snapshot of the master and slaves processes (m1 and s1).

In figure 4 we find the codes for m1 and s1 processes. From there we see that both master and slaves are sending (and receiving) the same data to the same process. Then, they are classified as master and slave types.

5 Conclusions

In this paper we have shown how KappaPi tool analyses the performance of a parallel program from the analysis of the efficiency of the execution shown at the trace file to the classification of the most important inefficiencies found. Lastly, we have introduced the possibility to use some user level information to look for some execution situations and analyse their performance.

In our opinion, there is a certain number of those execution situations that can be typical of a determinate parallel programming environment and these situations should be addressed in the same way.

References

- [1] Pancake, C. M., Simmons, M. L., Yan J. C.: Performance Evaluation Tools for Parallel and Distributed Systems. IEEE Computer, November 1995, vol. 28, p. 16-19.
- [2] Heath, M. T., Etheridge, J. A.: Visualizing the performance of parallel programs. IEEE Computer, November 1995, vol. 28, p. 21-28 .
- [3] Kohl, J.A. and Geist, G.A.: "*XPVM Users Guide*". Tech. Report. Oak Ridge National Laboratory, 1995.
- [4] Reed, D. A., Aydt , R. A., Noe , R. J., Roth, P. C., Shields, K. A., Schwartz , B. W. and Tavera, L. F .: Scalable Performance Analysis: The Pablo Performance Analysis Environment. Proceedings of Scalable Parallel Libraries Conference. IEEE Computer Society, 1993.
- [5] Reed, D. A., Giles, R. C., Catlett, C. E.. Distributed Data and Immersive Collaboration. Communications of the ACM. November 1997. Vol. 40, No 11. p. 39-48.
- [6] Hollingsworth, J. K., Miller, B, P. Dynamic Control of Performance Monitoring on Large Scale Parallel Systems. International Conference on Supercomputing (Tokyo, July 19-23, 1993).
- [7] Yan, Y. C., Sarukhai, S. R.: Analyzing parallel program performance using normalized performance indices and trace transformation techniques. Parallel Computing 22 (1996) 1215-1237.
- [8] Crovella, M.E. and LeBlanc, T. J. . The search for Lost Cycles: A New approach to parallel performance evaluation. TR479. The University of Rochester, Computer Science Department, Rochester, New York, December 1994.
- [9] Meira W. Jr. Modelling performance of parallel programs. TR859. Computer Science Department, University of Rochester, June 1995.
- [10] Fahringer T. , Automatic Performance Prediction of Parallel Programs. Kluwer Academic Publishers. 1996.
- [11] C. B. Stunkel, D. C. Rudolph, W. K. Fuchs, and D. A. Reed. Linear optimization: a case study in performance analysis. Proceedings of the fourth conference on Hypercube concurrent computers and applications, March 1989.
- [12] Geist, A. , Beguelin, A. , Dongarra, J. , Jiang, W. , Manchek, R. and Sunderam, V. , PVM: Parallel Virtual Machine, A User's Guide and Tutorial for Network Parallel Computing. MIT Press, Cambridge, MA, 1994.
- [13] Espinosa, A. , Margalef, T. and Luque, E. . Automatic Performance Evaluation of Parallel Programs. Proc. of the 6th EUROMICRO Workshop on Parallel and Distributed Processing, pp. 43-49. IEEE CS. 1998.

Extending PVM with consistent cut capabilities: Application Aspects and Implementation Strategies

Andrea Clematis¹, Vittoria Gianuzzi²

¹ IMA - CNR, Via De Marini 6, 16149 Genova, Italy,

Clematis@ima.ge.cnr.it,

WWW home page: <http://www.ima.ge.cnr.it>

² DISI, Università di Genova, Via Dodecaneso 35, 16146 Genova, Italy,

Gianuzzi@disi.unige.it,

WWW home page: <http://www.disi.unige.it/person/GianuzziV>

Abstract. Message passing libraries are now widely used to develop parallel and distributed applications. Despite different services are provided by the available packages, still little support is given for distributed consistent cut, a facility which constitutes the base for higher level services both at system level (e.g. distributed debugging and fault tolerance), and at the application level (e.g. distributed simulation). In this paper we discuss different strategies to integrate a consistent cut protocol in PVM system.

1 Introduction

The lack of access to the global state of a system is an important problem in distributed computing. Processes receive information by means of messages which, because of unpredictable delays, could give a non consistent view of the global situation, giving rise to non correct local decisions. In solving long-running problems, for example, the issue of how to handle the failures cannot be ignored. These applications must periodically store their global state, allowing the system to restart after a crash from the last saved state. Consistent cut allows to take a co-ordinated checkpoint, avoiding the need to maintain a log of sent messages. State saving operations have usually a dominant cost, however, it is important that the cut algorithm is reliable and simple. The same problem occurs in case of process migration for load balancing need. Consistent cut is also required to detect any stable property of a distributed system, like in a debugger system, or in a distributed simulation. In this case, cut operation could be repeated very often, and represents the dominant cost: also efficiency is then very important.

Chandy and Lamport [1] and Lai and Yang [10] have presented algorithms to compute a snapshot of a global system state, based on a consistent cut. Usually, the cut algorithm is implemented as embedded in system service or in the application. On the contrary, it could be implemented, once for all, in the communication library, providing the user with higher level communication primitives, optimised for the target architecture.

We report our experience in enhancing PVM message passing system with consistent cut capabilities. Two implementations have been developed in order to analyse different solutions and select the better alternative with respect to the following requirements:

- ★ to introduce a minimal and consistent extension of the application program interface of PVM;
- ★ to provide support for different applications and system services;
- ★ to introduce a minimal overhead in the number of exchanged messages and execution time;
- ★ to facilitate portability and integration of the cut service in new versions of PVM.

The two implementations are named CPVM and PVMsnap, and enhance the PVM 3.3 version with consistent cut capabilities [2, 7]. Before to continue, let us give a short definition of consistent cut, and two examples of its use.

Consistent cut: A *Global Cut* for a distributed program P is usually defined as the union of local histories for all processes in P , until some clock value, possibly different for each process. Then, the *cut event* is an internal event occurred at time $clock_k$ for each process p_k . An event occurred before its local cut event is said to be in the past of the cut, in the future otherwise.

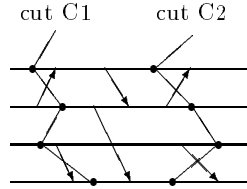


Fig.1: Non consistent (C1) and consistent (C2) cuts.

A cut is *Consistent* if between any two events e_i and e_j , respectively occurred in the future and in the past of the cut, the Happened Before relation is not established, that is $e_i \not\prec e_j$ (see Figure 1 for an example).

The Cut Algorithm allows to position the cut local event so that the cut results to be consistent. It is a total algorithm, that is, it is composed by a local part which defines the cut event and collects the messages in transit through the cut, and a global part which co-ordinates the local cuts.

A first example of use of the cut algorithm is to provide checkpoint services for a distributed Computation.

Global State Checkpoint: in case of process or node failure, the computation can be restarted from some previously saved intermediate state, corresponding to a consistent cut, and called global snapshot. Given a Consistent Cut C , a *Global Snapshot* for C is defined as: $\cup_{p_i \in P} S(p_i) \cup \cup_{ch_j \in C} M(ch_j)$ where $S(p_i)$ is the state of process p_i at its local cut event, and $M(ch_j)$ is the set of in-transit messages, that is, sent in the past and received in the future of the cut.

To deal with in-transit messages each process records its state upon the cut event, as well as a log of the following in-transit messages. Otherwise after the cut event, each process selectively receive only in-transit messages. The local checkpoint is performed after the receipt of all of them. Thus, the global snapshot includes only the process local state.

A second example of use is related with distributed simulation applications. **The Virtual Time synchronisation mechanism:** Virtual Time ([8]) is a general method to simulate synchronous systems on asynchronous ones, which can be used in various fields of distributed computation. In particular, it gives rise to the Time Warp simulation mechanism [9].

Each process maintains a local (virtual) clock, and sends time-stamped messages. Messages are assumed to arrive in the correct temporal order. If causality is violated, a roll-back operation is performed, and anti-messages are send, if needed, to annihilate messages sent out during the deleted time interval.

Uncontrolled diffusion of roll-back and unlimited extension of logs are avoided because the whole execution can be periodically committed, until a so called Global Virtual Time (GVT). GVT is defined, for a given consistent cut, as the minimum value among all local clocks and the time stamps of the in-transit messages.

In the following, a proposal about an extension of PVM - API with consistent cut functionality is presented. Then, the algorithm chosen for the cut and different implementation strategies inside PVM is discussed. Finally some experimental results and qualitative aspects of the two implementations are considered.

2 A PVM-API for consistent cut

Recalling the two application examples discussed above, each process involved in the computation needs to know the position of the cut event, the kind of the received messages (control or regular), their position with respect to the cut, and the occurrence of the local and global termination of the cut, in order to discard old checkpoints or to evaluate the GVT. The selective receipt of in-transit messages must be provided as well.

These are the minimal functionality which should be supported in both cases. The wish of making some type of operations more simple and efficient could suggest to augment this minimal set. Just for this reason in CPVM the API was richer than the minimal one, we will discuss it shortly in Section 4. Hereafter we present the PVMsnap API, which has been designed to satisfy the minimal changes requirement.

Since the cut algorithm is total, a *starter process* is identified: it starts the cut and executes other user-defined actions. All other processes will be referred as *receivers*. This distinction is only logical and has no impact on the API.

The starter process requires a cut by calling the special non blocking function `PVM_SNAP_STARTCUT()`. As a consequence, each process (starter included) is notified by PVM itself of the beginning of the cut interval. Each receiver is

also notified by PVM when no more in transit message have to be received. To distinguish whether the message is a regular (generated by the application) or control (PVM notification) one, the PVM receive function returned value may assume two new negative values: *PvmCutEvent* is returned if a new cut has begun, *PvmTermCut* notifies that no in-transit messages are still in transit.

To allow the user's program to know if a message is in transit, an additional return parameter (*transmsg*) has been provided to the PVM receive functions, with values `IN_TRANSIT` or `NOT_IN_TRANSIT`. As well as for the other parameters (*tid* and *msgtag*), selective receive is allowed also on this last argument. The user supplied value `ANY` matches both of the previous values.

The starter process receives its *PvmTermCut* message only after the global cut protocol termination, that is when all the in-transit messages through the cut have been received.

As an example of use of this interface, let us consider the receive procedure for the GVT evaluation. Part of the code is shown in Figure 2. The starter process, which does not participate on the simulation, starts the cut, receives the Local Virtual Time (LVT) from each process, computes the GVT and broadcasts such value. Each receiver (i.e. application process), at the end of the local cut, evaluates the LVT (supposing the first field of a message be its time stamp), sends it to the starter, and finally receives the GVT.

PVMsnap was also successfully integrated with UTCP (User Triggered Check-Pointing) [5, 3], a system for user controlled checkpoint and automatic rollback. parallel

The PVMsnap API is a minimal one and is based on a single new primitive, two new status code, and a new parameter for PVM receive functions.

3 Algorithms for consistent cut execution

Before addressing PVM implementation aspects let us summarise some features of algorithms for consistent cut.

The most known among the algorithms for performing a consistent cut, and probably the unique widely implemented, is also the first one proposed which does not require the freezing of the computation, that is, the Chandy- Lamport algorithm [1]. This algorithm sends a *marker* through every logical channel in order to ensure the consistency of the cut. In a network in which each process is allowed to communicate with each other, the number of markers is $O(n^2)$, where n is the number of processes. Such method has been used for example in CODINE [4], and in Fail-safe PVM [11].

In our implementations we select a different algorithm, presented by Lai and Yang [10], because of its easy implementation in PVM and its lowest complexity.

3.1 The Lai-Yang algorithm

In the Lai-Yang local cut algorithm, each computational phase (included between two consistent cuts) is painted by one out of two colours, white or red, and regular messages sent by a white (red) process are painted white (red).

The starter process

```

.....
info = pvm_startcut();
/* receipt of messages
   containing the LVT */
do
{
    bufid =
        pvm_rcv(ANY, ANY);
    switch(bufid)
    {
        case PvmCutEvent :
            GVT = MAX_VALUE;
            n=0; break;
        case PvmTermCut : break;
        default : /* reg. msg */
            pvm_upkint(&LVT,1,1);
            GVT = min(GVT, LVT);
            n++;
    }
}
while (n!=RecvNum);
pvm_initsend(PvmDataDefault);
pvm_pkint(GVT, 1, 1);
pvm_mcast(RecvTids,RecvNum,GVTtag);
.....

```

Receive instructions
for a receiver process

```

.....
transmsg=ANY;
bufid =
    pvm_rcv(ANY,ANY,&transmsg)
if (bufid >= 0) /* reg. msg */
{
    if (transmsg) /* in-transit */
    {
        pvm_upkint(&TimeStamp,1,1);
        LVT =
            min(CLOCK, Time_Stamp, LVT);
        /* appl.-specific code... */
    }
    else /* non in-transit */
    {
        /* appl.-specific code... */
    }
}
else
    switch(bufid)
    {
        case PvmCutEvent:
            LVT = MAX_VALUE; break;
        case PvmTermCut:
            pvm_initsend(PvmDataDefault);
            pvm_pkint(LVT,1,1);
            pvm_send(StarterId, LVTtag);
            break;
        default: /* error code... */
    };
.....

```

Fig.2: Code for the GVT evaluation

Initially, every process is coloured white. When a cut is required, the starter broadcasts a control message to all processes. Each white (red) process sets its own cut event when receiving such a control message, as well as just before receiving a regular message with a red (white) colour, and turns red (white). Then, all the white (red) messages received after the cut event are precisely messages in transit through the cut line.

A termination algorithm must be applied to detect the cut termination. Such a predicate can be evaluated, for example, by the starter, applying a deficiency counting termination detection algorithm, which requires each process to transmit to the starter the difference between the number of messages respectively sent and received until its cut event. Each process also sends to the starter an acknowledge for each in-transit message received. Since the starter knows the number of in-transit messages on the whole system, it can detect the global termination and broadcast the related control message. In this case, the message complexity is $O(n + M)$ where n is the number of processes and M is the number of in-transit messages.

Another termination detection algorithm can be used if each message is acknowledged. In this case, each process sends a control message to starter as soon

as all the messages it sent in the past of the cut have been acknowledged by the receiver process. The number of control messages falls to $O(n)$.

We may now better describe PVM implementation aspects.

4 Implementation strategies for consistent cut in PVM

and

The consistent cut protocol has been implemented in two versions, which differs for the API, the termination detection algorithm, and the implementation features.

CPVM [2] is an extension of PVM [6] library, which leaves the daemon completely unchanged. The deficiency counter termination algorithm has been used. Its API provides a set of functions slightly different from those presented in Section 2, being enriched of additional primitives useful to support the complete checkpoint/restart algorithm: for example, in-transit messages are not deleted from PVM buffers, as it usually happens in PVM when application programs unpack data carried by them, in order to minimise the time needed to record the checkpoint.

CPVM includes also an XPVM interface, slightly modified to visualise in different colours, regular, in-transit or not, and control messages.

In **PVMsnap** [3] the cut protocol has been embedded in both the PVM daemon and library: the major responsibility, that is the co-ordination of the cut with the starter and the cut termination detection, is ascribed to the daemon, while the PVM library send-receive functions perform the correct message delivery to the user. Process spawning and their exit from PVM are handled as well. Only inter-process communications handled by PVM daemon have been considered.

Let us consider some details of the PVMsnap implementation, which exhibits an higher level of difficulty with respect to CPVM, because of the involvement of the daemon.

4.1 PVMsnap: a tool for visible snapshot

PVM daemons use the connectionless UDP protocol, however, because of its unreliability, message acknowledgement is performed, in order to achieve higher reliability. Thus, each user message is acknowledged by the receiver PVM daemon, and this property has been used to detect the cut termination, reaching a message complexity of $O(n)$.

Upon starter request, the local daemon starts the protocol broadcasting a message to every daemon, itself included. Each daemon generates and multicasts a control message (*PvmStartCut*) to each local process, as soon as it receives the starter control message or a regular message painted with a different colour. Since communications between daemons and processes are managed in a FIFO style, each process will receive it before any other message whose sending event belongs to the future of the cut, independently of any selective receive performed

(control messages have the property to match every selective pattern). After the receipt of the *PvmStartCut* message, every outgoing messages will be painted with the new colour. Selective receive can be requested on possible in-transit messages.

Each daemon sends a control message to the starter when it has received all acknowledgements of user messages sent during the past cut interval. After the receipt of all these messages, the starter daemon broadcasts the final control message *PvmTermCut*.

Since messages must be painted with one out of two colours, a piggyback of 1 bit is required for every message: the additional bit is obtained from its tag field, without increasing the message size [2, 7, 12]. The colour is added when an output message is built, inside the PVM send functions, and it is filtered when received by the appropriate PVM functions, so that the user is not aware of it. The overhead in time is extremely small, as it will be shown in next Section.

5 Experimental results and comparative analysis

We have experimented both CPVM and PVMsnap on our workstation network.

The first data collected where aimed to assess the *send/receive overheads*, due to extra communication latency, introduced by each system on PVM applications which do not use the cut protocol. This overheads is almost null for the PVMsnap version (due to the daemon implementation), but it is very low also for CPVM, having a value lower than 0.5 percent for send operations and lower than 1.5 percent for receive operations.

The second data collected is aimed to analyse the duration of consistent cut in the two systems and their *overheads*. We consider three different types of application, each one composed of six processes and with each process running on a different workstation. The three applications are a network of distributed processes with a random communication pattern, a matrix multiplication on a regular grid, and Mandelbrot master - slave implementation. The communication pattern influences the cost of the cut, and its duration is longer for the distributed simulation and shorter for the matrix multiplication. The duration of the cut algorithm, measured on the starter process, is around 0,5 seconds for PVMsnap and around 1 second for CPVM. Considering that the execution of the cut is almost completely overlaped with normal computation and communication on the receiver process, this difference has no practical effect on application performances.

The overhead is in both cases less than 2 percent, considering a computation which last for around 100 minutes and executes the cut each minute. We have also some scalability data for PVMsnap. Using this system and considering a network of 60 processes we measured a duration of the cut algorithm of about 1 second, which indicates a good scalability of this implementation.

Besides efficiency it is important to consider other aspects like the user interface quality and the implementation cost.

The API quality is ensured by its consistency and minimal changes with respect to standard PVM. This is achieved by the PVMsnap which provides a general purpose service which can be easily embedded in different systems/applications.

For that it concerns the implementation the key problem is what to put in the daemon, and what in the library. In principle the daemon implementation should be more efficient. In our experience we measured a very low overheads in both solutions. On the contrary the daemon implementations may become less maintainable and portable on new version of PVM, thus introducing an higher cost. This is a key factor which makes the library based solution preferable.

Our next step will be to develop a version of PVMsnap based on library implementation.

In summary, the addition of cut capabilities to a message passing library is a feasible one which has a relatively low cost and does not impact on message system performances. It may offer interesting possibilities both at the application and system level, especially considering new directions of PVM, which seems to be the preferred tool for distributed co-ordination in metacomputing applications.

References

1. Chandy K., Lamport L.: Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Systems* **3**(1) (1985) 63–75
2. Clematis A., Gianuzzi V.: CPVM - Extending PVM for Consistent Checkpointing. *Euromicro Workshop on Parallel and Distributed Processing* (1996) 67–74
3. Clematis A., Deconinck G., Gianuzzi V.: A flexible state-saving library for message-passing systems. *6th Euromicro Workshop on Parallel and Distributed Processing* (1998)
4. <http://summa.physik.hu-berlin.de/rosen/codine-eng.html>
5. Deconinck G. et al. Fault Tolerance in Massively Parallel Systems. *Transputer Communications* **2**(4) (1994) 241–257
6. Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R., Sunderam R.: *PVM: Parallel Virtual Machine - A User Guide and Tutorial for Networked Parallel Computing*. The MIT Press Cambridge Ma. (1994)
7. V. Gianuzzi. An Optimal Snapshot Protocol for Acknowledged Message Passing, Tech. Rep. DISI-TR-96-6, 1996.
8. Jefferson D.R.: Virtual time. *ACM Trans. Programming Languages Syst.* **7**(3) (1985) 404–425
9. Jefferson D.R. et al.: Distributed simulation and the Time Warp Operating System. *12th SIGOPS* (1987) 77–93
10. Lai T.-H., Yang T.-H.: On distributed snapshots. *Inform. Process. Lett.* **25** (1987) 153–158
11. León J., Fisher A.L., Steenkiste P.: Fail-safe PVM: a portable package for distributed programming with transparent recovery. *CMU-CS-93-124* (1993)
12. Wisnüller R.: Debugging message passing programs using invisible message tags, in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, M. Bubak, J. Dongarra and J. Wasniewski (Eds.), LNCS 1332 (1997), (295–302)

Flattening on the Fly: efficient handling of MPI derived datatypes

Jesper Larsson Träff, Rolf Hempel, Hubert Ritzdorf, Falk Zimmermann

C & C Research Laboratories, NEC Europe Ltd.,
Rathausallee 10, D-53757 Sankt Augustin, Germany.
`traff@ccrl-nece.technopark.gmd.de`

Abstract. The Message Passing Interface (MPI) incorporates a mechanism for describing structured, non-contiguous memory layouts for use as communication buffers in MPI communication functions. The rationale behind the *derived datatype* mechanism is to alleviate the user from tedious packing and unpacking of non-consecutive data into contiguous communication buffers. Furthermore, the mechanism makes it possible to improve performance by saving on internal buffering. Apparently, current MPI implementations entail considerable performance penalties when working with derived datatypes. We describe a new method called *flattening on the fly* for the efficient handling of derived datatypes in MPI. The method aims at exploiting regularities in the memory layout described by the datatype as far as possible. In addition it considerably reduces the overhead for parsing the datatype. Flattening on the fly has been implemented and evaluated on an NEC SX-4 vector supercomputer. On the SX-4 flattening on the fly performs significantly better than previous methods, resulting in performance comparable to what the user can in the best case achieve by packing and unpacking data manually. Also on a PC cluster the method gives worthwhile improvements in cases that are not handled well by the conventional implementation.

1 Introduction

The mechanism for describing complicated, non-contiguous memory layouts provided by *derived datatypes* is an integral part of the Message Passing Interface (MPI) [5]. In MPI all communication functions take a datatype argument, which describes the structure of the communication buffer. This can alleviate the user from tedious packing and unpacking of structured data by hand, and saves memory in the application that would have been needed for intermediate communication buffers. Derived datatypes also play an important role for the MPI-2 IO functions [6]. Ideally, a high quality MPI implementation should perform at least as good when communicating non-contiguously stored data as packing and unpacking manually.

It is up to the MPI implementation to handle derived datatypes as efficiently as possible, for instance by communicating directly from the area of user memory determined by the datatype. In most MPI implementations, however, packing

into a consecutive internal communication buffer takes place, and is performed by MPI internal pack and unpack routines. This is the case for the portable MPICH implementation [2], and for the original NEC SX-4 MPI implementation, MPI/SX [4]. Apparently, the handling of derived datatypes in most current MPI implementations entail considerable overhead compared to what the user can achieve by doing packing and unpacking of structured data manually. A recent paper [3] discusses improvements by optimizing copy loops, and shows that this is in many cases beneficial. In this paper we propose what we think is a better method for handling derived datatypes, by exploiting regularities present in the memory layout specified by datatypes and repetition counts. One effect of *flattening on the fly* is that the overhead caused by parsing derived datatype descriptions is greatly reduced. More significantly, however, flattening on the fly makes it possible to utilize strided memory copy operations for all MPI datatypes, which for vector machines leads to large performance gains. The method has been integrated into the MPI/SX implementation, and is thus transparent to the user. On the SX-4 the resulting performance is comparable to, and sometimes better than what the user can in the best case achieve by packing and unpacking by hand. Flattening on the fly has also been tested on a PC cluster [1]. Worthwhile improvements have been achieved for cases that are not well handled by the MPICH implementation.

2 Derived datatypes in MPI

Derived datatypes is a means for describing layouts of data in memory. A memory layout is specified by a *type map*, which is a sequence of *primitive datatypes* and *displacements*. The primitive datatypes correspond to basic C or FORTRAN datatypes, such as integers (MPI_INT), floating point numbers (MPI_FLOAT, MPI_DOUBLE), characters (MPI_CHAR) etc. Arbitrarily complex memory layouts can be captured by combining primitive and already defined derived datatypes using the type constructor functions of MPI. A derived MPI datatype is characterized by its *size*, its *extent* and its constituent (derived) datatypes with associated displacements. The *size* of a derived datatype is the number of bytes taken up by the constituent datatypes. Its *extent* is the size of the memory segment “spanned” by the layout described by the datatype, that is the number of bytes from the beginning of the constituent datatype with the smallest displacement to the end of the constituent datatype with the largest displacement (for a more precise definition, consult [5, Section 3.2]).

Given an MPI (derived) datatype, a concrete area in memory, which will be referred to as a (*communication*) *buffer*, is designated by a *base address*, the type map of the datatype, and a *repetition count*. The concrete address of a primitive data element in the *i*th repetition of the datatype (with *i* between zero and the repetition count) can be computed by offsetting the base address with *i* times the type’s extent, and adding the displacement of the primitive element’s datatype.

A layout specified by a “structure” is illustrated in Figure 1. The structure in `userbuf` consists of blocks of varying size and type spaced at varying

offsets. The datatype is defined by calling the MPI type constructor function `MPI_Type_struct()` with arrays of block sizes, offsets and types for the blocks comprising the datatype. The example shows how this non-contiguous structure is packed into a contiguous communication buffer, `commbuf`. As can be seen,

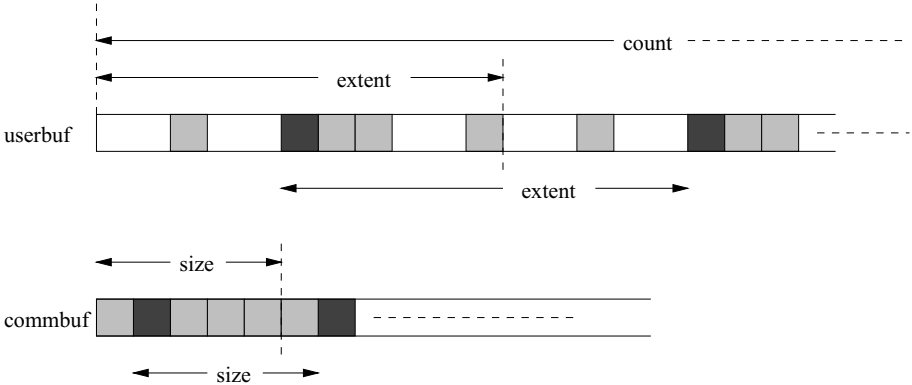


Fig. 1. Non-contiguous memory layout described by MPI Datatype (struct).

even if the layout specified by the datatype is irregular, the repetition count imposes regularity on the user buffer. The black areas, corresponding to a block of the structure, occur **extent** bytes apart as often as specified by the repetition count. In the packed communication buffer, the distance between the black areas is given by the **size** of the structure. Note that this still holds if the black area is itself a derived datatype. In this case the constituent types occur **extent** and **size** bytes apart in `userbuf` and `commbuf`. This kind of regularities can potentially make the copying from `userbuf` to `commbuf` more efficient.

The type map of a derived datatype can be succinctly represented as a datatype tree. Leaf nodes correspond to primitive datatypes and are characterized by their size in number of bytes. Internal nodes correspond to derived types, and are described by a repetition count (for vectors), their extent, size, and their children with associated displacements. The number (and type) of children depends on the type of the node. For instance, a structure or indexed type has as many children as there are components in the structure, while a vector has only one child.

The tree representation suggests how the type map can be reconstructed and used when packing an instance of a derived datatype to a communication buffer. By a simple recursive procedure the tree is traversed from root to leaves, computing the correct offsets in both user and communication buffers during the descent. When a leaf is reached copying of the leaf data takes place. The repetition count of each internal type node determines how many times this node's subtrees are visited. This method is implemented in MPICH, with some useful low-level optimization for the vector type [2, 3].

An obvious drawback of the method is that subtrees of the type tree are traversed many times. This is costly in the presence of large repetition counts as is often the case for vectors. This can be alleviated by *flattening* the datatype, i.e. by expanding the type map once. With an efficient representation of the type map as, say, a list, parsing of the datatype is no longer necessary. The mandatory call of `MPI_Type_commit()` before a new derived datatype can be used, provides a natural point for flattening to take place. However, (naive) flattening leads to a blow-up in the representation of the type map, which, in the presence of large repetition counts, can be prohibitive. A further disadvantage is that large scale regularities in the type layout, as illustrated by the example in Figure 1, are lost. Flattening does, however, give room for other kinds of nontrivial optimizations like fusion of consecutive blocks.

3 Flattening on the fly

Flattening is a tempting alternative for handling derived datatypes, since the recursive parsing of the type tree is replaced by a simple list traversal. Blow-up, and the fact that regularities in the memory layout are lost, makes this alternative less appealing. Procedure call overhead can be eliminated by converting the recursive formulation into an iterative equivalent, but this still has the drawback of losing regularities imposed by repetition counts. We introduce another method, called *flattening on the fly*, which, like flattening, has the advantage of eliminating the repeated traversal of the type tree, while still preserving regularities in the memory layout given by the repetition counts.

With flattening on the fly the datatype is parsed recursively, but the type tree is traversed only once. This is accomplished as follows. For each internal node in the type tree *one* recursive call is performed for each child; repetition count, extent, and size of the internal node are pushed on an explicit stack to be processed when a leaf is reached. The information on this stack completely determines how many times data of the leaf type has to be copied, where in memory data items appear (determined by the extents) and where they have to go in the communication buffer (determined by the sizes). For each of the MPI derived datatype constructors there is a corresponding rule for how count, extent and size are pushed onto the stack.

The following code fragment illustrates how a leaf is processed. Assume that a data structure for stack items has been defined, containing a repetition count `count`, a loop count `i`, a `size` field, an `extent` field, and a pointer to the previous stack element, `prev`. The loop count of each stack item is initialized to zero. The `top` pointer points to the current item, and is initialized to `stacktop`. Copying is done by counting through the repetition counts on the stack, increasing the addresses of the user and the communication buffer accordingly, in each iteration copying `leafsize` bytes from user to communication buffer. To find the buffer position for the next iteration we locate the item closest to the stack top with `i < count`. The loop count of this item is incremented, the positions in user and communication buffers incremented with the corresponding `extent` and `size`,

and all loop counts closer to the stack top are set to zero. The leaf is done when `i==count` for all stack items.

```
do {
    top = stacktop;
    if (top->i<top->count) memcpy(commbuf,userbuf,leafsize);
    while (top->i==top->count) {
        userbuf -= top->count*top->extent;
        commbuf -= top->count*top->size;
        top->i = 0;
        top = top->prev; /* pop stack */ if (top==NULL) break;
    }
    if (top!=NULL) {
        userbuf += top->extent;
        commbuf += top->size;
        top->i++;
    }
} while (top!=NULL)
```

Checks that copying is done within the boundaries of `userbuf` and `commbuf` can easily be incorporated into the code. Likewise, it is possible to start packing from an arbitrary point in the user buffer. This is required for a full-fledged integration into the MPI/SX implementation, and partly also for integration into MPICH.

Since all repetition counts are present on the stack when a leaf is reached, regularities due to large repetition counts can readily be exploited. Instead of a simple `memcpy()`, which copies just one data item, we pop one level off the stack, and do the copying for this level in one go with `count`, `extent` and `size` as arguments to a strided memory copy operation. On the SX-4 a highly efficient, vectorized, strided memory copy operation is available. To best exploit the strided memory copy operation, it is desirable to look for the stack level with the largest count, and do the copying from this level. We term this optimization “largest count first”. On the SX-4 this optimization is responsible for the good performance for derived datatypes which have a large repetition count close to the root of the datatype tree.

On some (scalar) architectures the straightforward implementation sketched here may result in bad cache behavior. Large extents or sizes at the copying level may cause cache misses in every iteration, resulting in an expensive copy loop. This can be alleviated by counting through the stack in a different order than sketched above, so as to stay within the cache as long as possible. On the SX-4, where vector-operations circumvent the cache, this is not an issue.

4 Experimental setup

We measure the communication performance with derived datatypes as the bandwidth obtained by sending messages of a given datatype containing a given number of items of some fixed base type. In the test one process sends messages of the derived data type, while the other receives data in a consecutive buffer. Only the send time is measured. We have defined different derived datatypes over

the same primitive base type, `MPI_DOUBLE`. For each datatype we exchange the same number of primitive data items, `items`. The user buffer is aligned to a word boundary. We estimate the overhead of working with MPI derived datatypes by comparing to the bandwidth that the user can achieve by manually packing the data into a contiguous buffer before sending. Results are given in Table 1.

On the SX-4 we compare the performance of the MPICH implementation [2], the current MPI/SX implementation [4], and the performance obtained with flattening on the fly. On the PC cluster flattening on the fly is compared only to the MPICH-based implementation. Bandwidths are averaged over 50 successive send operations.

We have experimented with the following datatypes:

Double: An array of type `double` with `items` doubles is sent as type `MPI_DOUBLE`.

This corresponds to the case where the user already has data in a contiguous buffer, and gives an upper bound on the communication performance.

Vector: A derived datatype created with the `MPI_Type_vector` type constructor. We look at three variants: (1) one vector with `items` doubles, (2) 100 vectors of `items/100` doubles, (3) `items/100` vectors of 100 doubles. All vectors have block length 1 and a stride of 11 doubles.

Nested vector: A vector of 10 items given as 10 blocks of one double with stride 2 is embedded in a vector with `items/100` items, count 1 and stride 2; 10 such vectors are sent.

Struct: A derived datatype created with the `MPI_Type_struct` constructor. The structure contains one double and has an extent of 11 doubles (as the simple vector types), which is enforced by having a second component of the structure define an upper bound using `MPI_UB`.

Indexed: A derived datatype created with the `MPI_Type_indexed` type constructor. We look at two variants: (1) an element contains 10 blocks, each with 10 doubles; `items/100` such elements are sent, (2) an element contains `items/100` blocks, each with 10 doubles; 10 such elements are sent.

Complex struct: A complex structure, consisting of doubles, contiguous, indexed, vector, and nested vector datatypes. We consider two variants: (1) the structure contains `items/10` items, and 10 such are sent, (2) the structure contains 10 items, and `items/10` such are sent.

For the SX-4 the performance achieved with flattening on the fly is in all cases comparable (to within a factor of at most 2) to what the user can at best achieve by packing and unpacking data by hand. In the cases of indexed(1) and complex structure(2), where user vectorization has been performed on inner loops, flattening on the fly gives significantly better results since it automatically exploits large surrounding repetition counts. This is very satisfactory for a general purpose method. The user can achieve similar performance only by writing a separate pack routine for each data layout, which is tedious for very irregular layouts. The importance of exploiting vectorization is clearly brought out by comparing the performance of the MPICH and MPI/SX implementations for the vector(1)-(3) cases. The MPI/SX implementation utilizes the vectorized, strided memory copy operation for MPI vector types, which leads to an improvement

		NEC SX-4				PC cluster		
Datatype	items	Bandwidths (mill. bytes/second)				Bandwidths		
		USER	MPICH	MPI/SX	FF	USER	MPICH	FF
Double	100	64.58	64.58	64.58	64.58	33.28	33.28	33.28
Vector(1)	100	64.97	27.57	41.11	34.55	25.73	26.18	20.48
Vector(2)	100	14.42	67.41	67.28	67.68	14.30	38.76	37.27
Vector(3)	100	16.11	27.85	41.67	35.19	17.01	27.06	21.15
Nested	100	15.67	26.11	30.36	26.78	14.06	27.55	19.67
Struct	100	64.93	2.84	4.70	30.93	25.73	5.05	17.85
Indexed(1)	100	64.82	25.50	32.08	19.84	24.08	22.54	21.79
Indexed(2)	100	64.72	24.07	29.61	33.05	23.68	22.94	26.71
Complex(1)	100	23.67	5.92	8.12	18.59	15.38	9.65	18.15
Complex(2)	100	44.31	5.92	8.11	18.59	23.94	9.58	18.29
Double	1000	640.51	640.51	640.51	640.51	66.73	66.73	66.73
Vector(1)	1000	570.03	47.04	316.45	265.49	43.54	47.54	37.55
Vector(2)	1000	132.32	49.65	101.70	210.42	30.14	48.56	36.34
Vector(3)	1000	149.62	47.45	276.04	229.98	30.46	47.71	37.58
Nested	1000	144.37	28.58	41.74	66.28	30.30	39.61	33.90
Struct	1000	616.28	3.03	5.24	252.78	43.50	5.58	28.76
Indexed(1)	1000	357.05	54.23	88.30	143.33	39.47	36.85	51.74
Indexed(2)	1000	333.85	54.23	88.33	143.36	38.67	36.91	51.75
Complex(1)	1000	206.02	34.49	54.62	93.80	30.95	30.93	43.99
Complex(2)	1000	60.88	6.92	10.04	163.66	38.60	11.50	37.59
Double	10000	3957.17	3957.17	3957.17	3957.17	62.34	62.34	62.34
Vector(1)	10000	2777.24	54.74	1769.26	1622.72	27.52	27.76	22.19
Vector(2)	10000	1162.64	54.96	911.13	771.45	22.14	28.12	22.06
Vector(3)	10000	1277.89	54.94	911.20	771.57	19.88	28.21	22.03
Nested	10000	1239.49	28.03	47.51	716.06	34.26	39.02	33.99
Struct	10000	2883.24	3.03	5.25	1700.33	27.98	5.18	19.41
Indexed(1)	10000	577.71	63.13	113.75	1186.74	48.26	42.74	59.02
Indexed(2)	10000	578.39	63.93	117.14	248.59	47.55	44.14	60.32
Complex(1)	10000	1593.52	97.39	514.13	847.24	34.65	39.77	49.65
Complex(2)	10000	58.06	6.98	10.35	1192.10	42.53	12.02	39.59
Double	100000	4297.06	4297.06	4297.06	4297.06	26.26	26.26	26.26
Vector(1)	100000	3516.53	30.26	3140.57	3052.88	13.56	12.30	11.78
Vector(2)	100000	3507.37	28.49	3140.44	3040.60	11.66	12.29	11.81
Vector(3)	100000	2007.34	28.80	1141.84	1826.65	10.30	13.28	10.27
Nested	100000	2620.33	19.03	26.25	2337.20	10.90	15.37	13.89
Struct	100000	3515.18	2.90	4.87	3102.83	12.76	4.34	10.98
Indexed(1)	100000	602.13	62.69	116.75	1957.49	15.70	16.19	15.23
Indexed(2)	100000	601.98	63.52	120.55	265.46	14.69	16.21	16.80
Complex(1)	100000	3367.33	121.53	2681.92	2857.19	13.60	15.54	13.71
Complex(2)	100000	57.53	6.24	8.79	1808.39	16.08	8.42	9.59

Table 1. Performance of derived datatypes on the SX-4 and the PC cluster. Column USER gives the performance achieved when packing manually. MPI/SX refers to the current MPI release for the SX-4. FF is flattening on the fly.

over the unvectorizable MPICH implementation of a factor 50-100 for 100000 items. The good performance of the MPI/SX implementation on simple vectors is lost on nested vectors and indexed and structured datatypes. Flattening on the fly is still able to exploit regularities due to large repetition counts, and achieves performance which is orders of magnitudes better than that of the other implementations.

On the PC cluster the improvements are less striking. For nested, structured and indexed types, worthwhile improvements of up to a factor 4 are achieved, although there is some extra overhead for simple vectors. For the instances with 100000 items, the bandwidth of the cluster drops [1], and effectively hides the differences between the two methods.

5 Conclusion

We presented a new method to improve the handling of MPI derived datatypes. *Flattening on the fly* achieves some of the same effects as flattening, but preserves regularities in the data layout which can be utilized in special, vectorizable copy routines. On the NEC SX-4 vector supercomputer significant performance improvements over previous implementations were achieved, also in cases of complex and nested data structures. With flattening on the fly, performance is comparable to what the user can in the best case achieve by manually packing and unpacking the data. On the SX-4 derived datatypes are handled efficiently enough that they should be the preferred alternative when dealing with complex data structures in applications using MPI.

Acknowledgment

Thanks to Maciej Gołębiewski for incorporating the new pack and unpack routines in the PC cluster MPI implementation.

References

1. M. Baum, M. Golebiewski, R. Hempel, and J. L. Träff. Dual-device MPI implementation for PC clusters with SMP nodes. In *Third MPI Developer's and User's Conference (MPIDC'99)*, pages 53–60, 1999.
2. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.
3. W. D. Gropp, E. Lusk, and D. Swider. Improving the performance of MPI derived datatypes. In *Third MPI Developer's and User's Conference (MPIDC'99)*, pages 25–30, 1999.
4. R. Hempel, H. Ritzdorf, and F. Zimmermann. Implementation of MPI on NEC's SX-4 multi-node architecture. *Future Generation Computer Systems*, 1999. To appear.
5. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998.
6. R. Thakur, W. Gropp, and E. Lusk. A case for using MPI's derived datatypes to improve I/O performance. In *Proceedings of SC98: High Performance Networking and Computing*. ACM/IEEE Press, 1998.

PVM Emulation in the Harness Metacomputing System: A Plug-In Based Approach

Mauro Migliardi and Vaidy Sunderam

Emory University, Dept. Of Math & Computer Science
Atlanta, GA, 30322, USA
om@mathcs.emory.edu

Abstract. Metacomputing frameworks have received renewed attention of late, fueled both by advances in hardware and networking, and by novel concepts such as computational grids. Harness is an experimental metacomputing system based upon the principle of dynamic reconfigurability not only in terms of the computers and networks that comprise the virtual machine, but also in the capabilities of the VM itself. These characteristics may be modified under user control via a "plug-in" mechanism that is the central feature of the system. In this paper we describe our preliminary experience in the design of PVM emulation by means of a set of plug-in.

1 Introduction

Harness [1] is a metacomputing framework that is based upon several experimental concepts, including dynamic reconfigurability and fluid, extensible, virtual machines. Harness is a joint project between Emory University, Oak Ridge National Lab, and the University of Tennessee, and is a follow on to PVM [2], a popular network-based distributed computing platform of the 1990's. The underlying motivation behind Harness is to develop a metacomputing platform for the next generation, incorporating the inherent capability to integrate new technologies as they evolve. The first motivation is an outcome of the perceived need in metacomputing systems to provide more functionality, flexibility, and performance, while the second is based upon a desire to allow the framework to respond rapidly to advances in hardware, networks, system software, and applications. Both motivations are, in some part, derived from our experiences with the PVM system, whose monolithic design implies that substantial re-engineering is required to extend its capabilities or to adapt it to new network or machine architectures.

Harness attempts to overcome the limited flexibility of traditional software systems by defining a simple but powerful architectural model based on the concept of a software backplane. The Harness model is one that consists primarily of a kernel that is configured, according to user or application requirements, by attaching "plug-in" modules that provide various services. Some plug-ins are provided as part of the Harness system, while others might be developed by individual users for special

situations, while yet other plug-ins might be obtained from third-party repositories. By configuring a Harness virtual machine using a suite of plug-ins appropriate to the particular hardware platform being used, the application being executed, and resource and time constraints, users are able to obtain functionality and performance that is well suited to their specific circumstances. Furthermore, since the Harness architecture is modular, plug-ins may be developed incrementally for emerging technologies such as faster networks or switches, new data compression algorithms or visualization methods, or resource allocation schemes – and these may be incorporated into the Harness system without requiring a major re-engineering effort.

The Harness project is in its initial stages, and we are in the process of defining the backplane architecture and developing the core framework. Simultaneously, we are also experimenting with a prototype kernel implementation as well as with plug-ins to realize different types of services. One major effort involves the creation of plug-ins that will emulate the PVM concurrent computing system within Harness. From the point of view of Harness research, this exercise will determine the viability of constructing new metacomputing facilities or even complete environments through the use of plug-ins. From a practical perspective, a suite of PVM plug-ins for Harness will provide a convenient and effective transition path for existing users and applications. In this paper, we describe the preliminary design of PVM emulation in Harness. We begin with an overview of the Harness model, describe an initial implementation and experiences with its use, and outline the proposed design for developing PVM plug-ins. A discussion of the status of Harness and the project outlook concludes the paper.

2 Architectural Overview of Harness

The fundamental abstraction in the Harness metacomputing framework is the **Distributed Virtual Machine (DVM)** (see figure 1, level 1). Any DVM is associated with a symbolic name that is unique in the Harness name space, but has no physical entities connected to it. **Heterogeneous Computational Resources** may enroll into a DVM (see figure 1, level 2) at any time, however at this level the DVM is not ready yet to accept requests from users. To get ready to interact with users and applications the heterogeneous computational resources enrolled in a DVM need to load **plug-ins** (see figure 1, level 3). A plug-in is a software component implementing a specific **service**. By loading plug-ins a DVM can build a consistent **service baseline** (see figure 1, level 4). Users may **reconfigure** the DVM at any time (see figure 1, level 4) both in terms of computational resources enrolled by having them **join** or **leave** the DVM and in terms of services available by **loading** and **unloading** plug-ins.

The main goal of the Harness metacomputing framework is to achieve the capability to enroll heterogeneous computational resources into a DVM and make them capable of delivering a consistent service baseline to users. This goal requires the programs building up the framework to be as portable as possible over an as large as possible selection of systems. The availability of services to heterogeneous computational resources derives from two different properties of the framework: the portability of plug-ins and the presence of multiple searchable plug-in repositories.

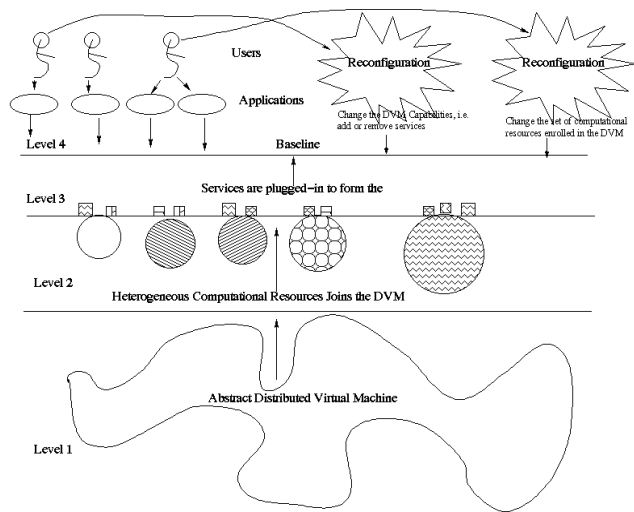


Figure 1 Abstract model of a Harness Distributed Virtual Machine

Harness implements these properties mainly leveraging two different features of Java technology. These features are the capability to layer a homogeneous architecture such as the Java Virtual Machine (JVM) [3] over a large set of heterogeneous computational resources, and the capability to customize the mechanism adopted to load and link new objects and libraries.

The adoption of the Java language has also given us the capability to tune the trade-off between portability and efficiency for the different components of the framework. This capability is extremely important, in fact, although portability at large is needed in all the components of the framework, it is possible to distinguish three different categories of components that requires different level of portability. The first category is represented by the components implementing the capability to manage the DVM status and load and unload services. We call these components **kernel level services**. These services require the highest achievable degree of portability, as a matter of fact they are necessary to enroll a computational resource into a DVM. The second category is represented by very commonly used services (e.g. a general, network independent, message-passing service or a generic event notification mechanism). We call these services **basic services**. Basic services should be generally available, but it is conceivable for some computational resources based on specialized architecture to lack them. The last category is represented by highly architecture specific services. These services include all those services that are inherently dependent on the specific characteristics of a computational resource (e.g. a low-level image processing service exploiting a SIMD co-processor, a message-passing service exploiting a specific network interface or any service that need architecture dependent optimization). We call these services **specialized services**. For this last category portability is a goal to strive for, but it is acceptable that they will be available only on small subsets of the available computational resources. These different requirements for portability and efficiency can optimally leverage the

capability to link together Java byte code and system dependent native code enabled by the Java Native Interface (JNI) [4]. The JNI allows to develop the parts of the framework that are most critical to efficient application execution in ANSI C language and to introduce into them the desired level of architecture dependent optimization at the cost of increased development effort.

The use of native code requires a different implementation of a service for each type of heterogeneous computational resource enrolled in the DVM. This fact implies a larger development effort. However, if a version of the plug-in for a specific architecture is available, the Harness metacomputing framework is able to fetch and load it in a user transparent fashion, thus users are screened from the necessity to control the set of architectures their application is currently running on. To achieve this result Harness leverages the capability of the JVM to let users redefine the mechanism used to retrieve and load both Java classes bytecode and native shared libraries. In fact, each DVM in the framework is able to search a set of plug-ins repositories for the desired library. This set of repositories is dynamically reconfigurable at run-time, users can add or delete repositories at any time.

3 The Design of a PVM Plug-In for a Harness DVM

Our initial approach to the problem of providing PVM compatibility in the Harness system has been guided by three objectives:

- requiring no changes into PVM applications to run in the new environment;
- minimizing the amount of changes to be inserted in the application side PVM library;
- achieving a modular design for the services provided by the PVM daemon.

We plan to achieve these goals by designing a set of plug-ins able to understand the original PVM library to PVM daemon protocol and to duplicate the services provided by the original PVM daemon. This approach allows us to provide complete compatibility with PVM legacy code while requiring only two changes in the PVM library on the application side:

- the adoption of internet domain sockets for the communication channel between the library and the daemon;
- the insertion of a Harness startup function.

Thus to run a legacy PVM application in the Harness PVM environment it is only necessary to link the original object code with the modified version of the library.

The services provided by the PVM daemons have been divided into five groups, namely process control, information, signaling, user-level message-passing and group operations. The services in each of these group are provided by a dedicated plug-in, namely the spawner plug-in, the database plug-in, the signal plug-in, the message-passing plug-in and the group plug-in. These plug-ins are loaded in the Harness kernels by the main PVMD plug-in when the Harness PVM environment is started or when an add-host command enrolls a new host. The main PVMD plug-in is also responsible of the loading of the plug-in that provides the message-passing service between the daemons. In figure 2 we show the actual sequence of the events in the

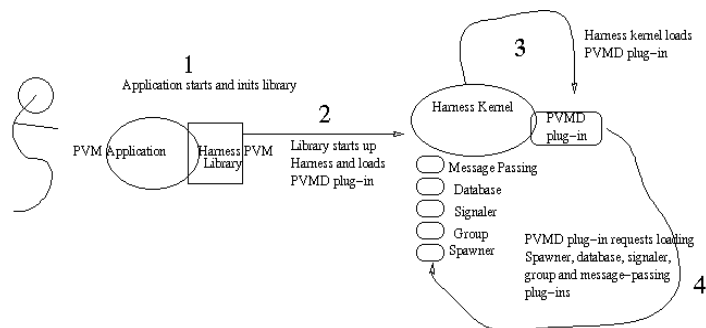


Figure 2 Sequence of events at Harness PVM startup

Harness PVM startup, while figure 3 shows the chain of events serving an `add_host` request. At PVM startup the PVM application (usually the console) starts up the PVM demon by issuing to the Harness kernel the command to load the main PVMD plug-in. This plug-in takes care to request the Harness kernel to load the services that are required to provide full PVM compatibility. When a task requests an `add-host` operation the local PVMD plug-in translates it into a request for the remote Harness kernel to load the main PVMD plug-in which then takes care of requesting the loading of the other needed plug-ins.

The first release of these plug-ins will provide only the services required to emulate PVM daemon's capabilities. However, the modularity of the design will easily let us substitute any plug-in with new versions in order to provide an enhanced version of the service. As an example, it will be easy to load a new version of the group plug-in

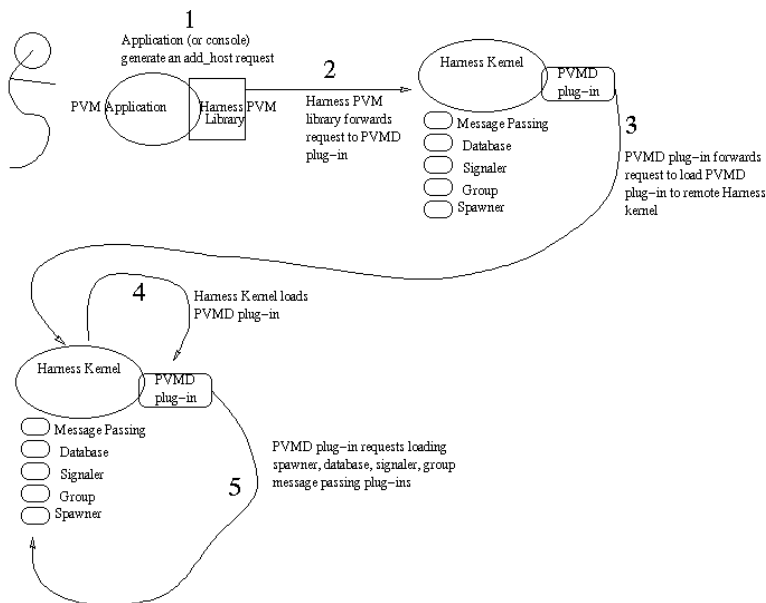


Figure 3 Sequence of events performed to service an `add_host` request.

in order to provide an extended set of group operations or a new database plug-in to provide an extended system querying capability. Besides, the Harness capability to hot swap services will allow run-time tuning of services to the set of hosts enrolled in the virtual machines, e.g. a specific version of message-passing plug-in could be loaded at run-time if a new communication fabric becomes available.

Besides fulfilling our primary goals this design has other advantages. The first one derives from the fact that the message-passing service provided by the message-passing plug-in needs only to peek at the destination field of a message in order to route it and does not need to know anything about the actual content of the message. This is beneficial for two reasons:

- the marshalling and un-marshalling of the data types is performed inside the Harness PVM library in C or Fortran thus we don't incur in the typical marshalling inefficiency due to the strong typedness of Java;
- it is extremely easy to substitute the message passing plug-in with another plug-in optimized to a specific communication fabric (be it a proprietary local network or an unreliable Internet connection) because they only need to move arrays of bytes.

Another benefit of our design is the fact that PVM applications can rely on the Harness capability to soft-install applications to move executable and libraries to the hosts in the VM. Thus it is not necessary to install the application and PVM itself on all the hosts of the VM, the Harness loader will do it as long as they are available on any host in the Harness DVM or on any one of the enlisted repositories.

A third, very important benefit of our design is the removal of the single point of failure represented by the master PVM daemon. In fact, providing PVM compatibility on top of the Harness system by means of a set of cooperating plug-ins, allowed us to exploit the Harness event subscription/notification service to implement a distributed control algorithm in the information management plug-ins. This algorithm is capable of reconstructing a consistent, up-to-date version of the PVM status after the crash of any daemon.

4 Related Works

Metacomputing frameworks have been popular for nearly a decade, when the advent of high-end workstations and ubiquitous networking enabled high performance concurrent computing in networked environments. PVM was one of the earliest systems to formulate the metacomputing concept in concrete virtual machine and programming-environment terms. PVM however, is inflexible in many respects that can be constraining to the next generation of metacomputing and collaborative applications. The Harness "plug-in" paradigm effectively alleviates these drawbacks while providing greatly expanded scope and substantial protection against both rigidity and obsolescence.

Jpvm [5] is a pure Java implementation of the PVM system. However it is not compatible with any traditional PVM application.

Legion [6] is a metacomputing system that began as an extension of the Mentat project. Legion can accommodate a heterogeneous mix of geographically distributed

high-performance machines and workstations. Legion is an object-oriented system where the focus is on providing transparent access to an enterprise-wide distributed computing framework. As such, it does not attempt to cater to changing needs and it is static in the types of computing models it supports as well as in implementation.

Globus [7] is a metacomputing infrastructure which is built upon the “Nexus” [8] multi-language communication framework. The Globus system is designed around the concept of a toolkit that consists of the pre-defined modules pertaining to communication, resource allocation, data, etc. However the assembly of these modules is not supposed to happen dynamically at run-time as in Harness. Besides, the modularity of Globus is at the metacomputing system level in the sense that modules affect the global composition of the metacomputing substrate.

Sun Microsystems Jini project [9] presents a model where a federation of Java enabled objects connected through a network can freely interact and deliver services to each other and to end users. In principle Jini shares with Harness many keywords and goals, such as services as building blocks and heterogeneity of service providers. However, Jini focuses on the capability to build up a world of plug-and-play consumer devices and, to cope with such a goal, increase the resolution of the computational resources that can be enrolled in a Jini federation. In fact in the Jini model these resources range from complete computational systems down to devices such as disks, printers, TVs and VCRs.

The above projects envision a model in which very high performance bricks are statically connected to build a larger system. One of the main idea of the Harness project is to trade some efficiency to gain enhanced global availability, upgradability and resilience to failures by dynamically connecting, disconnecting and reconfiguring heterogeneous components. Harness is also seen as a research tool for exploring pluggability and dynamic adaptability within DVMs as the design of the PVMD plug-in shows.

5 Conclusions

In this paper we have described our plug-in based design for providing PVM emulation in the Harness metacomputing system together with its rationale and advantages. Our design allows achieving object level compatibility with PVM legacy application while giving PVM user access to the new features provided by the Harness system. Besides it will provide to PVM users a smooth transition path to a programming environment that allows combining different programming paradigms.

Harness it's still in its initial stage, however our prototype is capable of:

- defining services in term of abstract Java interfaces in order to have them updated in a user transparent manner;
- adapting to changing user needs by adding new services to heterogeneous computational resources via the plug-in mechanism;

The design of our Harness PVM emulation shows that it is feasible to define distributed computing environment in terms of plug-ins. This approach allows the incorporation of new features and technological enhancements into existing systems such as PVM without loosing compatibility with legacy applications.

References

- 1 M. Migliardi, V. Sunderam, A. Geist, J. Dongarra, Dynamic Reconfiguration and Virtual Machine Management in the Harness Metacomputing System, Proc. of ISCOPE98, pp. 127-134, Santa Fe', New Mexico (USA), December 8-11, 1998.
- 2 A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Mancheck and V. Sunderam, PVM: Parallel Virtual Machine a User's Guide and Tutorial for Networked Parallel Computing, MIT Press, Cambridge, MA, 1994.
- 3 T. Lindholm and F. Yellin, The Java Virtual Machine Specification, Addison Wesley, 1997.
- 4 S. Liang, The Java Native Interface: Programming Guide and Reference, Addison Wesley, 1998.
- 5 A. J. Ferrari, Jpvm, Technical report, <http://www.cs.virginia.edu/ajf2j/jpvm>
- 6 A. Grimshaw, W. Wulf, J. French, A. Weaver and P. Reynolds. Legion: the next logical step toward a nationwide virtual computer, Technical Report CS-94-21, University of Virginia, 1994.
- 7 I. Foster and C. Kesselman, Globus: a Metacomputing Infrastructure Toolkit, International Journal of Supercomputing Application, May 1997.
- 8 I. Foster, C. Kesselman and S. Tuecke, The Nexus Approach to Integrating Multithreading and Communication, Journal of Parallel and Distributed Computing, 37:70-82, 1996
- 9 Sun Microsystems, Jini Architecture Overview, available on line at <http://java.sun.com/products/jini/whitepapers/architectureoverview.pdf>, 1998.

Implementing MPI-2 Extended Collective Operations

Pedro Silva and João Gabriel Silva

Dependable Systems Group, Dept. de Engenharia Informática,
Universidade de Coimbra, Portugal
ptavares@dsg.dei.uc.pt, jgabriel@dei.uc.pt

Abstract. This paper describes a first approach to implement MPI-2's Extended Collective Operations. We aimed to ascertain the feasibility and effectiveness of such a project based on existing algorithms. The focus is on the intercommunicator operations, as these represent the main extension of the MPI-2 standard on the MPI-1 collective operations. We expose the algorithms, key features and some performance results. The implementation was done on top of WMPI and honors the MPICH layering, therefore the algorithms can be easily incorporated into other MPICH based MPI implementations.

1 Introduction

MPI-2 brings major extensions to MPI: dynamic processes, one-sided communications and I/O [1]. Down the line of the new features come the extensions to the collective operations. These extensions open the doors to collective operations based on intercommunicators, bring a user-option to reduce memory movement and also two new operations: a generic alltoall and an exclusive scan.

The intercommunicator represents a communication domain composed of two disjoint groups, whereas the intracommunicator allows only processes in the same group to communicate [2]. Thus, due to its structure, the intercommunicator supports client-server and pipelined programs naturally. MPI-1 intercommunicators enabled users to perform only point to point operations. MPI-2 brings the generalization of the collective operations to intercommunicators which further empowers users in moving data between disjoint groups. The intercommunicator and the operations it supports are also significant because of the dynamic processes. Solely the intercommunicator enables the user to establish communication between groups of processes that are spawned or joined [3]. It thus expands beyond the bounds of each MPI_COMM_WORLD, unlike the intracommunicators.

This paper embodies a first approach to implement the MPI-2's Extended Collective Operations. We present algorithms for the intercommunicator based collective operations which draw from the existing MPICH [4] intracommunicator algorithms. The focus is on the intercommunicator based operations, as these represent the main extension of the MPI-2 standard on MPI-1's collective operations. We also present some performance results. The objective of this work was to ascertain the feasibility and effectiveness of implementing the operations based on the existing collective algorithms.

There have been many research projects on MPI collective operations. Most of these projects concentrate on specific characteristics of the computation environment. Some focus on the interconnection hardware, such as the LAMP project which set its sights on a Myrinet [5]. Other on the interconnection topology: The collective algorithms for WANs, a project conducted by Kielmann et al. [6], or the MPI-CCL library project that assumed a shared medium [7]. Yet others, like [8] and [9], provide algorithms that attain efficient collectives on heterogeneous NOWs. However, there are no projects that cover the collective operations based on intercommunicators, to the best of our knowledge.

We will show that a simple implementation is feasible using the algorithms from the existing collective operations based on intracommunicators. We also present some performance figures to show that the intercommunicator functions have a good performance and scale as well as the intracommunicator counterparts. This work represents a first step into incorporating the extensions that MPI-2 brings to collective operations on WMPI[10][11]. WMPI is a full implementation of MPI-1 for SMPs or clusters of Win32 platforms. Due to the fact that the top layers of WMPI are based on MPICH, the implementation of the extended collective operations can be easily added to MPICH or others alike.

2 Algorithms

This section lays out the algorithms of the intercommunicator operations. The algorithms are based on existing intracommunicator algorithms because these are simple. First, we present the communication patterns of the rooted operations, which fall into two categories: One-to-all where one process contributes data to all other processes; all-to-one, where all processes contribute data to just one process. Second, we present the algorithms for the all-to-all operations, where all processes contribute data and all processes receive data. Finally we detail the barrier operation.

2.1 Rooted Operations

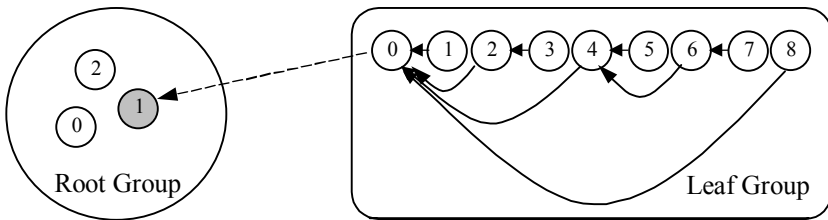


Fig. 1 – Intercommunicator Reduce (12 processes)

`MPI_Reduce` involves a reduce operation (such as sum, max, logical OR) with data provided by all the members of the leaf group, whose result is stored at the user-defined root process. The implementation of the `MPI_Reduce` operation draws heavily from its homonymous intracommunicator operation. The reduction is made up

of a local reduce (i.e. with the local group's intracommunicator) in the leaf group and a send from the leaf group's leader to the user defined root process. Figure 1 displays the message passing involved in the intercommunicator reduction operation. The processes to which the user passes `MPI_PROC_NULL` in the *root* argument (represented in Fig. 1 by the processes ranked 0 and 2 in the root group), call no message passing routines. These are completely passive in the operation and return immediately. This applies to the other rooted operations as well.

`MPI_Bcast` involves the passing of data from the user defined root to all processes in the leaf group. The `MPI_Bcast` function is also based on a recursive splitting tree algorithm, as the reduce. The root sends the user-provided data to the remote group's leader.

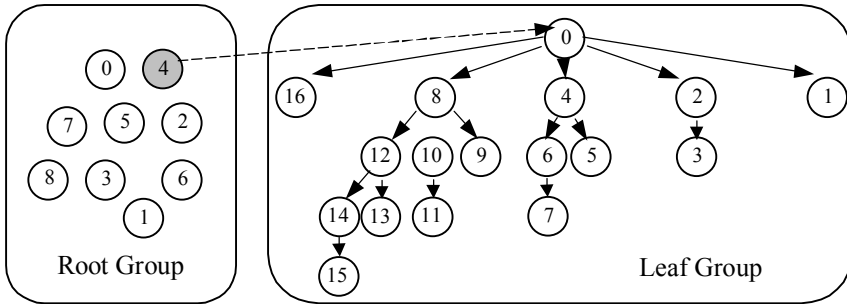


Fig. 2. – Intercommunicator Broadcast (26 processes)

At the leaf group the data is broadcast via a local intracommunicator based broadcast. Figure 2 displays the message passing that each process performs; process with local rank 4, on the left, is the user defined root. The local broadcast, as the local reduce in `MPI_Reduce`, have $O(\log N)$ time complexity, where N is the number of processes.

`MPI_Gather` has the user defined root collect data from all the processes in the leaf group. The `MPI_Gather(v)` operations are based on a linear algorithm that is similar to its intracommunicator counterpart. The user defined root process posts immediate receives to obtain the data from each process in the leaf group. In turn, each of these processes sends one message to the root process. This is a direct generalization from the intracommunicator gather operations.

`MPI_Scatter` has all processes in the leaf group provide data to the user defined root process. The `MPI_Scatter(v)` operations also have a linear implementation. The root process sends a distinct chunk of data to each of the processes in the leaf group. The root passes the messages via immediate sends to all processes and then waits for these to finish. The recipients simply call a blocking receive. As the aforementioned operations this one draws directly from the existing intracommunicator based counterparts. The scatter and gather operations have $O(N)$ algorithms.

2.2 All-to-All Operations

The MPI-2 standard implies that the MPI_Reduce_scatter should be thought of as an intercommunicator reduce, followed by an intracommunicator scatter. However, at least for a direct implementation and using the current reduce and scatter algorithms, an intra reduce followed by an inter scatterv is generally less costly. The first involves passing the results (obtained from the reduce operation) from each group to the other and only then does the leader process perform the scatter. On the other hand, the latter involves sending the results directly to the final recipient of the remote group. Thus there is one less hop in passing the results through the communication medium. In order for the inter scatterv to take place, the leaders must obtain the *recvcounts* argument from the other side. The fact that the latter algorithm involves this exchange is of no real consequence because it will probably be less heavy than transferring the results and, mainly, because the exchange is overlapped with the reduction operation. The leaders exchange the *recvcounts* and only then do they perform the reduce operation, thus taking advantage of the fact that they are the last to intervene in the local reduction operation. As for the inter scattervs, one for each group, these also lap over because they are based on non-blocking operations.

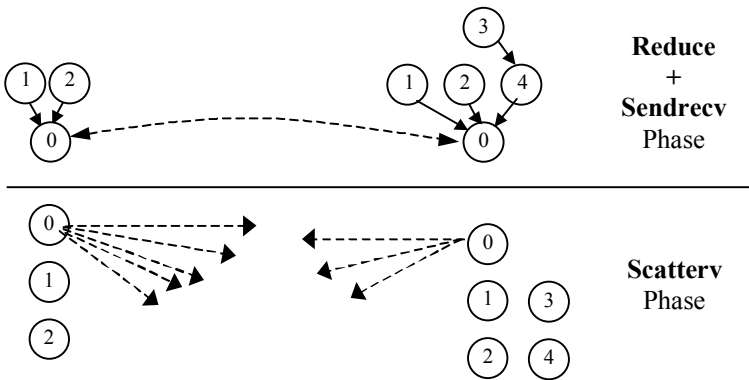


Fig. 3. – Intercommunicator Reduce_scatter (8 processes)

The MPI_Reduce_scatter operation was thus implemented as an intracommunicator reduce, followed by two intercommunicator scattervs. Figure 3 shows the three communication constituents of the operation in two phases – the exchange of *recvcounts* between the leaders, the simultaneous reduce on each group, and finally the scatter of data from the leaders to each member of the leaf group.

MPI_Allgather has each group provide a concatenation of data items from every member process to the other group. The MPI_Allgather and MPI_Allgatherv may be implemented in a straightforward manner which has two phases. First, every process from each group sends the data to the leader of the other group. The leader waits for the corresponding immediate receives to conclude and then broadcasts the data it received to the processes in its local group. The remainder of the processes in each local group join the leader in the broadcast. This function is actually very different

from its intracommunicator counterpart which is based on MPI_Sendrecv between neighboring processes.

MPI_Alltoall has each process from group A provide a data item to every process in group B, and vice versa. The MPI_Alltoall(v)(w) family of operations are composed of immediate receives and sends between each pair of processes from opposing groups. In an attempt to diminish the contention on the communication infrastructure, one group posts the receives first and then the sends while the other first posts the sends and only then posts the receives. When the intercommunicator is constructed a variable is set that identifies which group is elected to go first.

MPI_Allreduce has the result of the reduction from group A spread to all members of group B, and vice versa. The MPI_Allreduce operation is a three phase operation. First each group participates in an intracommunicator reduce. Second, the first elected root group broadcasts the results to the remote group. Third, the groups perform another inter broadcast where the roles are reversed. This way the reduction operation happens simultaneously on both groups. Also, the broadcast may be overlapped as the processes responsible for the distribution of data (i.e. the roots of each broadcast tree) are also the ones that return earlier from the first inter broadcast.

2.3 MPI_Barrier

MPI_Barrier is a synchronization operation in which all process in group A may exit the barrier when all of the processes in group B have entered the barrier, and vice versa. This operation is based on a hypercube, just as the intracommunicator barrier. To implement the algorithm based on a hypercube, the two groups that make up the intercommunicator are merged; one group keeps its local ranks while the other groups' ranks are shifted (we add the remote's size). In order to use the algorithm based on a hypercube one only has to translate the rank and check whether to use the intracommunicator or the intercommunicator. This way, we kept the algorithm simple through a direct generalization of the intracommunicator barrier.

3 Performance Tests

This section presents test results of some of the intercommunicator operations. To help quantify the results, these are juxtaposed with results obtained with the intracommunicator operations under the same conditions. To better compare the operations, the intercommunicator based operations that are rooted have only one process in the root group. This way the intercommunicator operation is closer to that of the intracommunicator operation; it avoids MPI_PROC_NULL processes in the root group. As for the all-to-all operations, the intercommunicator is divided evenly, each group has four processes. The testbed consisted of eight Windows NT 4.0 workstations (Pentium II based with 128MBytes of RAM), interconnected with a 100Mbps switch. These served as the remote communication testbed. The tests on shared memory were performed on a dual Pentium Pro NT 4.0 machine.

3.1 MPI_Reduce

Most of the intercommunicator collective operations perform as well as their intra counterparts. The tests on the MPI_Reduce, summarized in Figure 4, show that reduce operation is one such case. The inter reduction takes about the same amount of time as the intra one, on average over all processes.

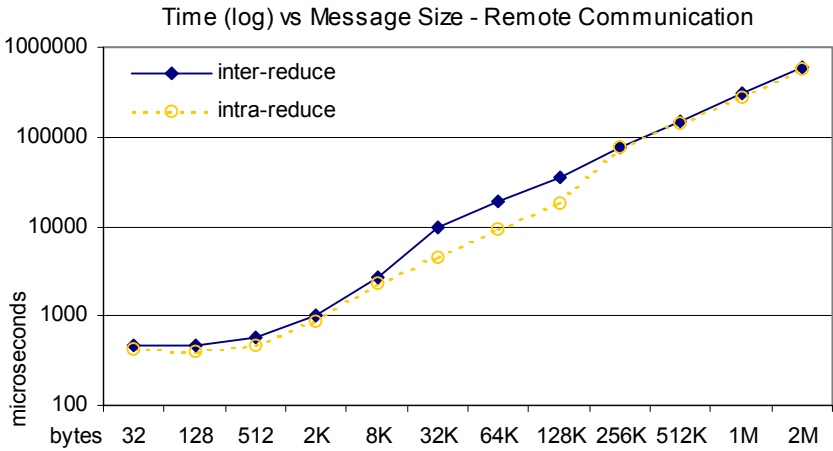


Fig. 4. – Intercommunicator and intracommunicator Reduce on 8 processes

A closer look at the execution time for each process revealed that the leaf processes, which only send data, take about the same amount of time on intra and inter operations. The processes that are truly responsible for the difference, small as it is, are the user defined root and the leaf group’s leader which perform the last receive and send, respectively. The same tests in shared memory also resulted in similar execution times between the intra and inter reduce.

3.2 MPI_Reduce_scatter

Table 1 shows the results of MPI_Reduce_scatter via remote communication for a 2MB message size.

Size (bytes)	Execution time of each participant process (microseconds)							
	0	1	2	3	4	5	6	7
2097152	1445076	1435866	1439482	1416597	1471134	1439796	1443290	1391354

Table 1. Intercommunicator Reduce_scatter on 8 processes via remote communication

The results show that the inter scatters overlap well since all processes have a similar execution time. Also, as expected, the leaders (ranked 0 and 4) take longer to finish as these are responsible for scattering the results to the processes of the remote group. The intra reduce scatter execution times are above the ones for the inter, for instance, for a 2MB of data it takes on average, over all processes, 1803691

microseconds. This is due to the fact that all processes participate actively in both the reduction and the scatter of data while in the inter only half perform each operation.

3.3 MPI_Allgather

In MPI_Allgather's case, the inter execution times are well below those of the intra allgather. The allgather results are evidence of just how different the intra operation is from the inter allgather. This is due mainly to the fact that the intra allgather uses blocking send and receives while the inter allgather uses non-blocking routines to take advantage of the overtaking rule in MPI. This way the leader may receive from each process out of order. The intra version cannot use non-blocking sends and receives because each process must first receive the data before forwarding the data to its neighbor. In distributed memory, the execution times were quite further apart as one pays a high price for passing messages over the relatively slow network. The test results for the MPI_Allgather operation are presented in Figure 5.

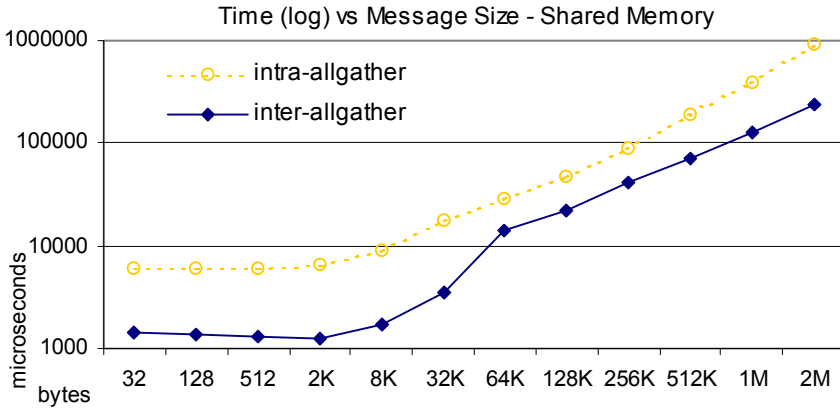


Fig. 5. – Inter and intracommunicator allgather on 8 processes via shared memory

4 Conclusions and Future Work

In this paper we presented a first step towards incorporating the MPI-2 collective operations' extensions into WMPI. We illustrated algorithms for intercommunicators based on the existing intracommunicator collective operations from MPICH. The implementation strives to avoid using the always expensive message passing (especially in networks) and also data movement within each process. We conclude that the operations can easily be implemented and have a good performance in fairly homogeneous environments. Furthermore, the operations have been tested and will provide a useful communication infrastructure for the implementation of the rest of MPI-2 on WMPI.

The next step for the collective operations is to provide algorithms for heterogeneous environments. NOWs, which are commonly heterogeneous, are becoming more and more attractive for cost-effective parallel computing. In order to truly take advantage of such an environment, one needs to incorporate some form of adaptability into the collective algorithms. Specifically, if one considers networked systems that are composed of SMPs, the bandwidth alone is astronomically different between processes on one machine and processes on distinct machines. ECO [8] and [9], mentioned above, already have shown that adaptability is computationally worthwhile. One way of attaining such adaptability, with low performance cost, is to analyze the computing environment before the computation. ECO uses such a strategy. ECO's library comes with an offline tool which tries to guess the network topology by analyzing the latencies between every pair of computers. In most cases, the tool produces communication patterns that result in efficient collective operations.

A similar approach might work for WMPI, however, due to the fact that MPI-2 brings dynamic processes, this tool cannot be offline. One way to circumvent the problem is to analyze the interconnections between processes every time a communicator is built and include a distribution tree, or another auxiliary structure, in the communicator. With respect to building the communication trees without compromising the performance, [9] proposes two approaches. The Speed-Partitioned Ordered Chain and the Fastest Node First approaches are of low complexity and generate communication patterns for fast and scalable collective operations.

Bibliography

1. MPI Forum – <http://www.mpi-forum.org>
2. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The Complete Reference, Vol. 1 – The MPI Core* (2nd Edition). MIT Press, 1998.
3. W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference, Vol.2 - The MPI Extensions*. MIT Press, 1998.
4. MPICH – <http://www-unix.mcs.anl.gov/mpi/mpich/>
5. Local Area Multiprocessor at the C&C Research Laboratories, NEC Europe Ltd. <http://www.ccr1-nece.technopark.gmd.de/~maciej/LAMP.html>
6. T. Kielmann, R. Hofman, H. Bal, A. Plaaat, and R. Bhoedjang. *MagPie: MPI's Collective Communication Operations for Clustered Wide Area Systems*. In Symposium on Principles and Practice of Parallel Programming, Atlanta, GA, May 1999.
7. Jehoshua Bruck, Danny Dolev, Ching-Tien Ho, Marcel-Catalin Rosu, and Ray Strong. *Efficient Message Passing Interface (MPI) for Parallel Computing on Clusters of Workstations*, 7th Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA-95 Santa Barbara, California, July 1995.
8. B. Lowekamp and A. Beguelin. *ECO: Efficient Collective Operations for Communication on Heterogeneous Networks*. In International Parallel Processing Symposium, pages 399-405, Honolulu, HI, 1996.
9. M. Banikazemi, V. Moorthy, and D. Panda. *Efficient Collective Communication on Heterogeneous Networks of Workstations*. In International Conference on Parallel Processing, pages 460-467, Minneapolis, MN, August 1998.
10. J. Marinho and J.G. Silva. *WMPI – Message Passing Interface for Win32 Clusters*. In Proc. of the 5th European PVM/MPI Users' Group Meeting, pp. 113-120, September 1998.
11. WMPI – <http://dsg.dei.uc.pt/wmpi>

Modeling MPI Collective Communications on the AP3000 Multicomputer

Juan Touriño and Ramón Doallo

Dep. of Electronics and Systems, University of A Coruña, Spain
{juan,doallo}@udc.es

Abstract. The performance of the communication primitives of a parallel computer is critical for the overall system performance. The performance of the message-passing routines does not only depend on the hardware of the communication subsystem, but also on their implementation. When users migrate parallel codes to a multicomputer, they are unaware of the quality of the implementation and they can find that the performance of their parallel applications makes worse. In this work, we evaluate some representative collective communication routines of the MPI message-passing library: broadcast, scatter/gather, total exchange and reduction operations. Specifically, experimental results were conducted on the Fujitsu AP3000 multicomputer using Fujitsu MPI/AP version 1.0, in order to derive models for these MPI routines.

1 Introduction

The Fujitsu AP3000 is a distributed-memory multiprocessor. The nodes (UltraSparc-II processors at 300 Mhz) are connected via a high-speed communication network (AP-Net) in a two-dimensional torus topology. The AP3000 architecture is described in detail in [5]. Currently, it can be programmed by using message-passing libraries (MPI, PVM or Fujitsu APLib, a proprietary library) or a data-parallel language (an HPF implementation of the Portland Group). We have selected MPI for our experiments because it is the library most widely used by the AP programmers.

A complete set of measures can be done to assess quantitatively the performance of the communication subsystem, by using appropriate tools and benchmark suites, such as the NAS Parallel Benchmarks [1] or the Parkbench suite [6]. We have focused on low-level tests to characterize basic MPI collective communication primitives (that is, communication patterns that involve all the processors in a communicator): broadcast, scatter/gather, total exchange and reduction. These primitives are the basis for the design of more complex communication patterns in a parallel application. Our aim is to estimate communication overheads with simple models, which can help AP application developers to design or migrate parallel programs.

There are performance results for different parallel computers in the literature but, at the present time, we have not found performance models for MPI primitives on the Fujitsu AP3000. This work is organized as follows: in the next

subsection, we introduce well-known models and their associated parameters to characterize communication overheads. In Section 2, we derive models for the MPI collective communication primitives on the AP3000 and experimental results are presented to confirm the accuracy of the models. Finally, conclusions are discussed in Section 3.

1.1 Communication Modeling Concepts

In point-to-point communication, message latency (T) can be modeled as an affine function of the message length n (in bytes):

$$T(n) = t_s + t_b n \quad (1)$$

where t_s is the startup time and t_b is the transfer time per data unit. Distance is not a factor when considering latency between any two processors, due to the wormhole routing on the AP3000.

Hockney [3] proposes the following model to characterize point-to-point communications:

$$T(n) = \frac{n_{\frac{1}{2}} + n}{Bw_{as}} \quad (2)$$

where Bw_{as} is the asymptotic bandwidth, that is, the maximum throughput achievable when n approaches infinity and $n_{\frac{1}{2}}$ is the half-peak length, the message length required to obtain half of Bw_{as} . The parameters of Equations 1 and 2 are related as follows: $Bw_{as} = 1/t_b$ and $n_{\frac{1}{2}} = t_s/t_b$.

We re-interpret $n_{\frac{1}{2}}$ as the message length for which t_s and $t_b n$ have the same weight in the total message latency. For $n > n_{\frac{1}{2}}$, the transfer time has a greater percentage of T than t_s , and for $n < n_{\frac{1}{2}}$, t_s dominates T . Following this interpretation, we could similarly define the parameters $n_{\frac{1}{4}}$ (n such that $t_s = 3T(n)/4$), $n_{\frac{3}{4}}$ (n such that $t_s = T(n)/4$) ... and use them to classify the set of messages according to the weight that t_s and $t_b n$ represent in T . We consider that the qualitative terms "short message", "long message", "very long message" ... are machine-dependent and should be numerically established according to the quantitative delimitation of these parameters.

In a previous work [7] we have estimated the parameters of Equations 1 and 2 for MPI blocking point-to-point communication on the AP3000:

$$t_s = 69\mu s, \quad t_b = 0.0162\mu s, \quad n_{\frac{1}{2}} = 4260 \text{ bytes}, \quad Bw_{as} = 58.87 \text{ Mbytes/s}$$

The parameters t_s and t_b were also estimated for PVM and APLib. We obtained for the transfer time the same result as in MPI, but lower values for the startup time: $53\mu s$ in PVM/AP and $46\mu s$ in APLib.

Collective communications can be characterized by the model proposed by Xu and Hwang [8], which is a generalization of the point-to-point model:

$$T(n, p) = t_s(p) + \frac{n}{Bw_{as}(p)} \quad (3)$$

Latency is a function of two variables, n and the number of processors p . $Bw_{as}(p)$ can also be expressed as $1/t_b(p)$. The half-peak length for collective communications is defined as $n_{\frac{1}{2}}(p) = t_s(p)/t_b(p)$.

2 Collective Communications

In order to derive expressions for collective communication overhead, we have performed a wide set of tests combining different message sizes (from 0 bytes up to 1 Mbyte) and number of processors (from 2 up to 12 processors, the maximum available in our system). Cache warm-up effects and timing outliers were removed to obtain accurate measures. In each test, the processors are firstly synchronized using a barrier (**MPI_Barrier**). As each test consists of hundreds of iterations, a barrier is included to avoid a pipelined effect, that is, some processors may start the next call to the collective primitive even before all processors finish the current collective routine. Besides, the barrier also prevents the network contention that may appear by the overlap of collective communications executed on different iterations of the test. Both effects distort the latency measurements.

2.1 Broadcast

In a broadcast (**MPI_Bcast**) a source processor (root) sends a message to a group of destination processors. After the experimental tests, the fitting of the parameters of Equation 3 led to the following results:

$$t_s(p) = 69 \log_2 p \mu s, \quad t_b(p) = 0.0162 \log_2 p \mu s$$

$$n_{\frac{1}{2}}(p) = 4260 \text{ bytes}, \quad Bw_{as}(p) = \frac{61.73k}{\log_2 p} \text{ Mbytes/s}$$

where $k=10^6/2^{20}$ is a constant introduced to consider the base 2 versus base 10 discrepancy.

It can be observed that the parameters of the broadcast model for $p=2$ are the same as the ones of the point-to-point model (see Subsection 1.1). Therefore, the expression for the latency of **MPI_Send/Recv** can be considered as a particular case of the **MPI_Bcast** expression for $p=2$. Besides, the parameter $n_{\frac{1}{2}}(p)$ is a constant, independent of p .

As was to be expected, the complexity of the broadcast is $O(\log_2 p)$, which shows that it was implemented using a tree-structured approach. The hardware of the Fujitsu AP1000 (the predecessor of the AP3000) included a specific network, the B-net (Broadcast network) [4], for one-to-all communications. Using this network, broadcast routines (and other collective primitives) with latency independent of p (that is, $O(1)$) could be implemented. In [7] we found that the latency of the broadcast in PVM/AP and APLib is $O(p)$ on the AP3000; therefore, they are inefficient implementations.

2.2 Scatter/Gather

A scatter/gather is a collective communication in which a fixed root processor sends/receives a distinct collection of data to/from every other processor.

In a scatter (**MPI_Scatter**), a data structure that is stored on a single processor is distributed across the processors. The communication overhead on the AP3000 is modeled as:

$$t_s(p) = 58 + 53p \mu s, \quad t_b(p) = 0.0209(\log_2 p)^{-0.3830} \mu s$$

$$n_{\frac{1}{2}}(p) = \frac{58 + 53p}{0.0209(\log_2 p)^{-0.3830}} \text{ bytes}, \quad Bw_{as}(p) = 47.80k(\log_2 p)^{0.3830} \text{ Mbytes/s}$$

In a gather (**MPI_Gather**), a distributed data structure is collected onto a single processor. The parameters obtained for this primitive are:

$$t_s(p) = \frac{1}{0.0135 - 0.0030 \log_2 p} \mu s, \quad t_b(p) = 0.0252(\log_2 p)^{-0.1473} \mu s$$

$$n_{\frac{1}{2}}(p) = \frac{39.66(\log_2 p)^{0.1473}}{0.0135 - 0.0030 \log_2 p} \text{ bytes}, \quad Bw_{as}(p) = 39.66k(\log_2 p)^{0.1473} \text{ Mbytes/s}$$

where the message length n of Equation 3 is the number of bytes of the message to be scattered from the root processor (in **MPI_Scatter**), and the total number of bytes of the message gathered in the root processor (in **MPI_Gather**).

In both primitives, $t_b(p)$ is a decreasing function, and it tends to be constant as p increases. It can be due to the fact that, as p increases, the size of the message to be sent/received by each processor is smaller and these communications can be overlapped through the torus interconnection network. Nevertheless, the startup increases with the number of processors and is specially high in the scatter operation because the root processor has to split the message among the processors.

2.3 Total Exchange

In this collective communication each processor sends a distinct collection of data to every processor. It can be viewed as a series of scatters or as a series of gathers. MPI provides this operation through the **MPI_Alltoall** routine. The following values were derived from the curve fitting:

$$t_s(p) = 89p \mu s, \quad t_b(p) = 0.0291 + 10^{-4} 8.84p \mu s$$

$$n_{\frac{1}{2}}(p) = \frac{89p}{0.0291 + 10^{-4} 8.84p} \text{ bytes}, \quad Bw_{as}(p) = \frac{k}{0.0291 + 10^{-4} 8.84p} \text{ Mbytes/s}$$

The startup time is $O(p)$ and is the highest of the collective routines under evaluation. Although the transfer time is also $O(p)$, the slope increases very slowly and it is almost constant.

2.4 Reduction

In a reduction operation (`MPI_Reduce`) each processor contains an operand, and all of them are combined using a binary operator that is successively applied to each one. The results were obtained for an `MPI_Sum` reduction of `MPI_DOUBLE` data (1 double=8 bytes in our system):

$$t_s(p) = 90 \log_2 p - 15 \mu s, \quad t_b(p) = 0.0222 \log_2 p \mu s$$

$$n_{\frac{1}{2}}(p) = 4054 - \frac{676}{\log_2 p} \text{ bytes}, \quad Bw_{as}(p) = \frac{45.05k}{\log_2 p} \text{ Mbytes/s}$$

These values may vary by selecting a different operator and/or data type. The MPI reduction is, as the broadcast, $O(\log_2 p)$ (tree-structured). As n and p increase, the reduction becomes the collective operation with the highest latencies. We must take into account that this routine involves many additional computations.

2.5 Experimental and Estimated Results

Figures 1-4 show an overview of the latencies of the MPI collective communications on the AP3000. The filled symbols represent the estimated values of the latencies, following the derived models, whereas the dotted symbols are the experimental values of the latencies. In many cases the estimated values are hidden by the measured values, which means a good modeling. We must remark that the regressions are for the interval $p=2-12$ and the results cannot be extrapolated for $p > 12$ (perhaps the values of the parameters and even the type of fit could change).

Figures 1 and 2 show latencies for different message sizes in an 8-processor configuration. In Figures 3 and 4 the message size is set for 16 bytes and 256 Kbytes, respectively, and the number of processors varies from 2 up to 12. Figure 3 focuses on short-message latency (dominated by the startup time) and Figure 4 on long-message performance (dominated by the transfer time). Many of the comments of the previous subsections can be observed graphically.

3 Conclusions

The characterization of the communication overhead is very important to estimate the global performance of the parallel application and to detect possible bottlenecks. Besides, taking into account the results reported, users should not only choose a multicomputer where to run their parallel applications, but also select the most adequate message-passing library to program the AP3000, by comparing the MPI library with other libraries (PVM and APLib), as discussed in [7]. Machine vendors should provide the parameters of these models (or, at least, complexities) for the basic communication routines.

The models presented in this work have a reasonable accuracy, although they work worse for medium-size messages. The total exchange operation presents the

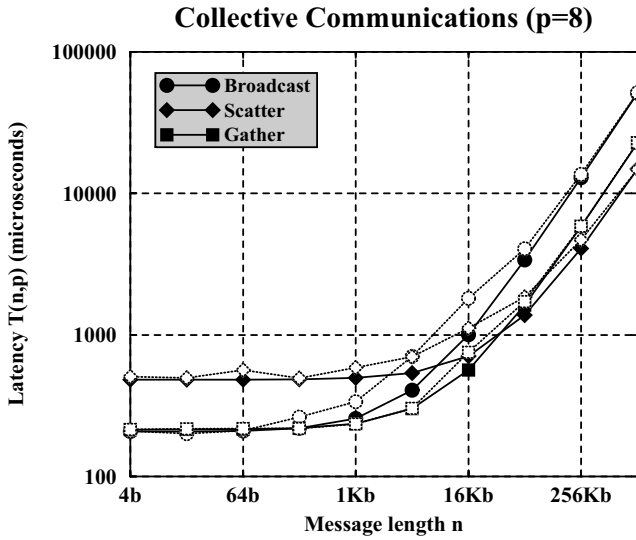


Fig. 1. Measured and estimated latencies over 8 processors for various message sizes (broadcast, scatter and gather)

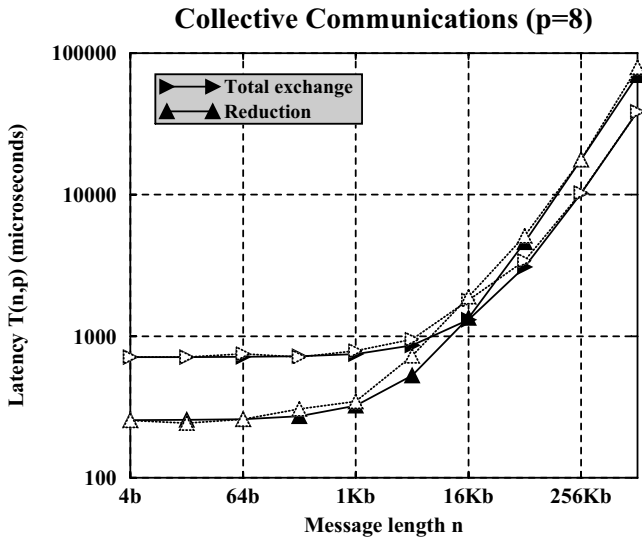


Fig. 2. Measured and estimated latencies over 8 processors for various message sizes (total exchange and reduction)

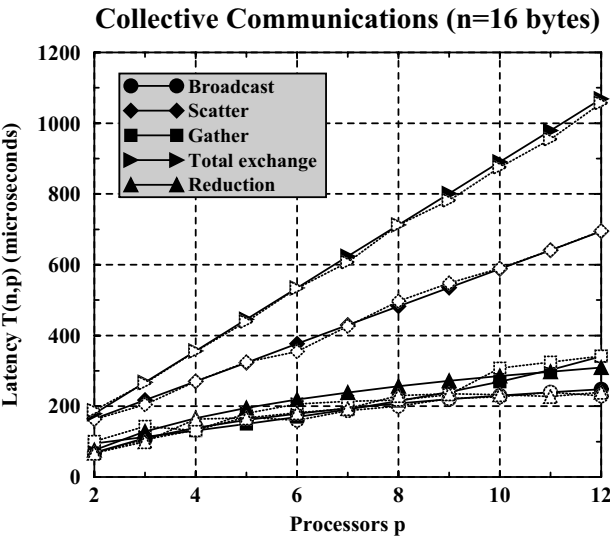


Fig. 3. Measured and estimated latencies for various machine sizes using 16-byte messages

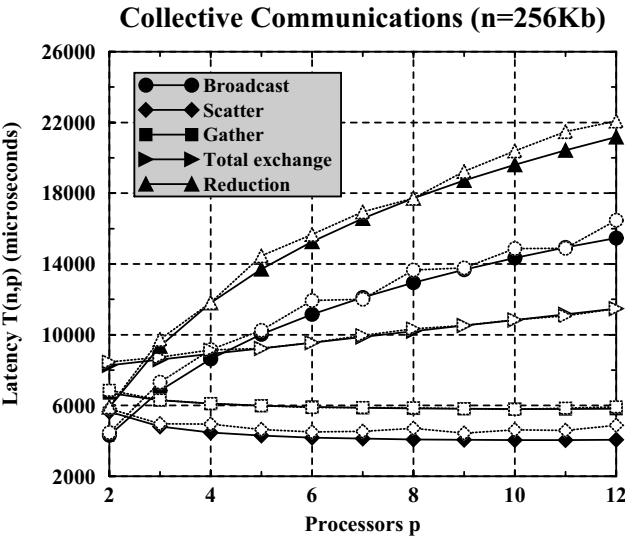


Fig. 4. Measured and estimated latencies for various machine sizes using 256-Kbyte messages

highest startup, whereas the reduction routine has the highest transfer time of the collective primitives under evaluation. We consider that, taking into account the underlying communication hardware of the AP3000 [5], the latencies of the MPI primitives could be greatly improved. In fact, MPI/AP was constructed using too many communication layers, which results in high latencies.

As future work, we intend to characterize more MPI collective communications, such as `MPI_Allreduce`, `MPI_Allgather`, `MPI_Reduce_scatter` and parallel prefix operations (`MPI_Scan`).

Acknowledgments

The tests were performed using the Fujitsu AP3000 located at the Galician Supercomputing Center, CESGA (Santiago de Compostela, Spain). This work was supported by the CICYT of the Spanish Ministry of Education under contract TIC96-1125-C03, and by the European Union, project 1FD97-0118-C02.

References

1. Bailey, D., Harris, T., Saphir, W., van der Wijngaart, R., Woo, A., Yarrow, M.: The NAS Parallel Benchmarks 2.0. Tech. Report NAS-95-020, NASA Ames Research Center (1995) (Release 2.3 available at <http://science.nas.nasa.gov/NAS/NPB>)
2. Dongarra, J., Dunigan, T.: Message-passing Performance of Various Computers. Tech. Report UT-CS-95-299, Computer Science Dept., University of Tennessee, Knoxville (1995)
3. Hockney, R.W.: The Communication Challenge for MPP: Intel Paragon and Meiko CS-2, *Parallel Computing* **20**(3) (1994) 389–398
4. Ishihata, H., Horie, T., Shimizu, T.: Architecture for the AP1000 Highly Parallel Computer, *Fujitsu Sci. Tech. J.* **29**(1) (1993) 6–14
5. Ishihata, H., Takahashi, M., Sato, H.: Hardware of AP3000 Scalar Parallel Server, *Fujitsu Sci. Tech. J.* **33**(1) (1997) 24–30
6. Parkbench Committee: Public International Benchmarks for Parallel Computers, *Scientific Programming* **3**(2) (1994) 101–146 (Release 2.1.1 available at <http://www.netlib.org/parkbench>)
7. Touriño, J., Doallo, R.: Performance Evaluation and Modeling of the Fujitsu AP3000 Message-Passing Libraries. To appear in EUROPAR'99, Toulouse, France (1999)
8. Xu, Z., Hwang, K.: Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2, *IEEE Parallel & Distributed Technology* **4**(1) (1996) 9–23

*MPL**: Efficient Record/Replay of nondeterministic features of message passing libraries

J. Chassin de Kergommeaux¹, M. Ronsse², and K. De Bosschere²

¹ ID-IMAG, B.P. 53, F-38041 Grenoble Cedex 9, France

² ELIS, Universiteit Gent, St.-Pietersnieuwstraat 41, B-9000 Gent, Belgium

Abstract. A major source of problems when debugging message passing programs is the nondeterministic behavior of the promiscuous receive and nonblocking test operations. This prohibits the use of cyclic debugging techniques because the intrusion caused by a debugger is often large enough to change the order in which processes interact. This paper describes the solutions we propose to efficiently record and replay the nondeterministic features of message passing libraries (MPL) like MPI or PVM. It turns out that for promiscuous receive operations it is sufficient to keep track of the sender of the message, and for nonblocking test-operations to keep track of the number of failed tests. The proposed solutions have been implemented for an existing MPI-library, and performance measurements reveal that the time overhead of both record and replay executions is very low with respect to the (nondeterministic) original execution while the size of the log files remains very small.

1 Introduction

A program is called nondeterministic if two subsequent runs with identical user input cannot be guaranteed to have the same behavior. A program can be nondeterministic, even in the case when two runs produce exactly the same output.

Although nondeterminism is often a desired feature in dynamically evolving systems (e.g., for load balancing, scheduling, etc.), it makes debugging very cumbersome, especially in parallel and distributed programs. Indeed, the fact that one process is slowed down by a debugger will most likely change the order in which processes interact, and can eventually change the complete execution path. Hereby, symptoms of programming errors might suddenly appear, disappear or change.

Hence, unless we succeed in making nondeterministic program executions deterministic, cyclic debugging techniques cannot be used with this class of programs. A well-known technique to make a nondeterministic program deterministic, is to observe an execution (e.g., by recording the outcome of all nondeterministic choices made by the program), and to impose this trace on subsequent executions (with the same input). The replayed execution is now completely deterministic, and can be used any number of times to debug the program.

Debugging an erroneous program then amounts to recording an erroneous execution and to applying cyclic debugging techniques during subsequent replayed (erroneous) executions. In order for this technique to be effective (i) the perturbation resulting from the recording operation ought to be kept sufficiently low so that errors appearing during non-recorded executions do not vanish in recorded ones and vice-versa, and (ii) the overhead during replay should be acceptable.

The record/replay technique can be implemented in two different ways: either by tracing the message data, and by imposing these data to the receive and test operations during replay (*contents driven replay*), or by tracing the order of racing events, and by imposing that order during replay (*control driven replay*). Instant Replay [10] is an example of control driven replay for shared memory programs. It is obvious that contents based replay requires trace files that are at least an order of magnitude bigger than required for control based replay. Therefore, control driven replay is a better candidate for an efficient implementation of the record phase.

On the other hand, control driven replay only works in the assumption that the replayed executions depart from the same initial state and that the next state in each process p_i can be derived from the previous state, and a possible input (either user input, or an incoming message). Furthermore, all outgoing messages must be derivable from the program state that produces it. In other words, the program must comply with the rules of causality, and given identical user input, the contents of the messages must be guaranteed to be identical if the same control is imposed on the execution. For message passing systems compliant with these assumptions, a trace file with just a description of the nondeterministic choices made by the program is sufficient to guarantee a correct replay.

Most execution replay systems are based on the Instant Replay mechanism which was initially designed for shared memory parallel programming models [10], but it was later adapted to message passing models [11], to an asynchronous distributed programming model [7], and to a fairly complex object-oriented distributed programming model [8].

The only system that comes close to our work is the NOPE system [9] that was designed to deterministically replay MPI programs. In contrast to our work, the NOPE system considers promiscuous receive operations as the only source of nondeterminism. It does not deal with the nondeterminism caused by nonblocking test operations. This is however important: any program that depends on the number of test-operations performed (e.g., for time-outs) cannot be correctly replayed.

In the literature on record/replay, a great deal of attention is given to the reduction of the trace size because this might be the factor that makes record/replay not feasible for long-running programs. A substantial reduction of the trace size could be obtained by using vector clocks [14] (for message passing systems), Lamport clocks [15] (for shared memory systems), and the properties of the programming model [5]. Our work goes beyond all the attempts for the compression of the trace data that originates from the test-functions. We successfully aggregate a succession of failed test-events into a counter and a final

test-event. To the best of our knowledge, aggregating events has never been used as a technique for reducing trace sizes.

In this paper we start with an overview of the nondeterministic features of message passing libraries, followed by a description of the solutions we propose to efficiently deal with them during the record phase and the replay phase. We then continue with an effective trace compression scheme based on the aggregation of nonblocking test-functions. This scheme not only reduces the size of the traces, but also speeds up the replay considerably.

2 Nondeterministic features of message passing libraries

We assume that messages in a point-to-point communication comply with the non-overtaking property which means that successive messages sent between two nodes are always received in the order in which they were sent (this is the case for MPI and PVM). Hence, since messages are assumed to be produced deterministically, receive operations in a private point-to-point communication that specify the sender are also deterministic.

The non-overtaking property does however not hold for messages that originate from different senders. Therefore, the so-called promiscuous receive operations (e.g., *MPI_Recv(...,MPI_ANY_SOURCE,...)*) which can receive a message from any other node are nondeterministic which means that the order in which communication events are taking place can differ between two executions, leading to different execution paths (see figure 1). In a sense, they play the role of ‘racing’ store operations in nondeterministic programs on multiprocessors.

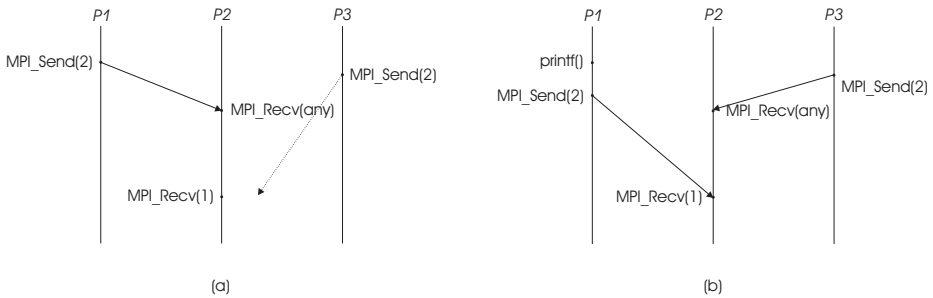


Fig. 1. The result of a promiscuous receive operation can depend on small timing variations.

Besides the promiscuous receive operations, there is another class of instructions that can cause nondeterminism in a message passing program:¹ nonblocking

¹ This paper only deals with nondeterminism due to the parallel nature of a program. Sources of nondeterminism that are also present in sequential programs such as *random()* or external input are not treated in this article.

test functions. This class of functions has been overlooked in previous papers on record/replay for message passing libraries [9].

Nonblocking test operations are however intensively used in message passing programs, e.g., to maximally overlap communication with computation: a nonblocking receive operation returns a request object as soon as the communication is initiated, without waiting for its completion. The request objects can be used to check the completion of the nonblocking operations by means of test-operations, which can in turn be blocking (*Wait*), or nonblocking (*Test*).

By the very fact that the test operations are nonblocking, they can be used in polling loops. The actual number of calls will depend on timing variations of parallel program, and is thus non-deterministic. Although many programs will not base their operations on the number of failed tests, some could do so (e.g., to implement a kind of time-out), and hence cannot be correctly replayed when not recorded (see Figure 2).

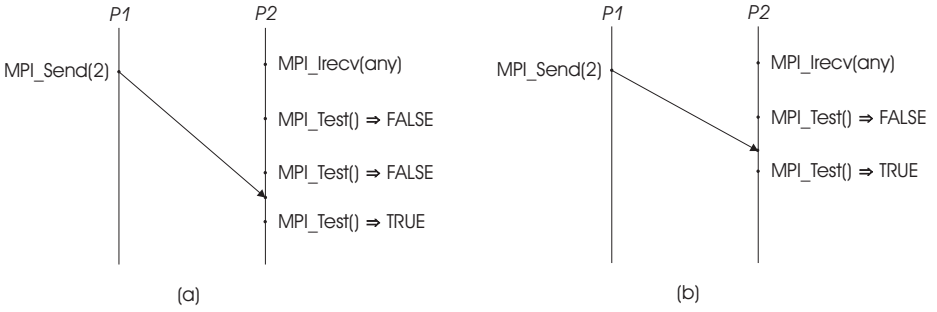


Fig. 2. The number of nonblocking test operations can depend on small timing variations.

A similar situation may occur for series of calls to functions testing for the arrival of a message (*MPI_IProbe*).

3 Execution replay for promiscuous receive and nonblocking test operations

As mentioned above, it is possible to use control driven replay for replaying message passing programs: it suffices to make sure that all nondeterministic choices are made in exactly the same way. In this paper, we focus on promiscuous receive operations and nonblocking test operations.

For the promiscuous receive operations, it suffices to record the actual sender of the messages received. This information can then be used during replay to force the promiscuous receive operations to wait on a messages originating from a particular sender process. Hence, a promiscuous receive operation can be made deterministic during replay by replacing it on-the-fly by a point-to-point receive

operation. Notice that this does not mean that the promiscuous receive operation is statically replaced by a point-to-point receive operation. Instead, every individual call to the promiscuous receive is dynamically replaced by the corresponding point-to-point receive.

For the nonblocking test operations, control driven replay implies that the setting of the condition that is tested for (e.g., the arrival of a message), must be postponed until the required number of test-operations is carried out. Only then, the operation that sets the condition can be allowed to resume. Replaying these test functions is no problem for a pure control driven replay system, where the order of *all* message operations is logged.

This approach has however a serious drawback. Nonblocking test operations are typically used in polling loops where they can in theory run for a very long time. Every iteration of the polling loop will add a bit of information to the trace file, although the polling loop is actually used to wait on a particular event. The fact that the trace file grows while the process is just waiting is not acceptable. It turns out that contents driven replay is actually more efficient for test functions than control driven replay. This is because a test function normally reports a series of failures, followed by a success unless the program finished before the test succeeds, when the request is cancelled (`MPI_Cancel`) or when the application stops polling and uses an `MPI_Wait` to wait for the completion of the request. Since all test functions (for one request) are identical we can count them and log this single number. During the replay phase, this number is decremented for each executed test function and as soon as the number becomes zero, we return `TRUE` and force an `MPI_Wait`. For an unsuccessful series of test functions, we log a number that is bigger than the number of executed test functions. This will force all test functions to fail during the replay phase as the counter never becomes zero. A similar solution was adopted to record the number of unsuccessful calls to `MPI_IProbe`.

Special care needs to be taken so that the number of unsuccessful tests is read from the traces as soon as the first `MPI_Test` of a series is performed. Similarly, in case of a non-blocking promiscuous receive operation `MPI_IRecv`, followed by a series of tests, the identification of the sender node needs to be read from the traces at the time the receive operation is called. These two problems were addressed by assigning request numbers to the first `MPI_Test` of a series as well as to each promiscuous `MPI_IRecv` call. These request numbers are stored in the trace files with the number of unsuccessful tests (resp. sender node identification) and the trace files are sorted before the first replayed execution. More technical details on the solution may be found in [4].

This approach dramatically reduces the trace size of test operations, especially in applications that use the nonblocking operations intensively.

Using contents based replay for the nonblocking test operations has yet another unexpected advantage. Since we do know the result of the failing test operations, there is actually no more need to call the actual message passing library routines anymore. We can now on-the-fly replace the failing test-operations by a call to a stub that just reports the failure, and replace the successful test

operation by the corresponding blocking test (*Wait*) operation. This only works if we can guarantee that the failing nonblocking test operations do not cause side effects, which is the case for MPI and PVM.

Replacing the message passing library routines by stubs has a dramatic effect on the replay speed of the nonblocking test-functions. Programs that do a lot polling on request objects might execute quite a bit faster during replay because the message passing library is not called anymore (see figure 3).

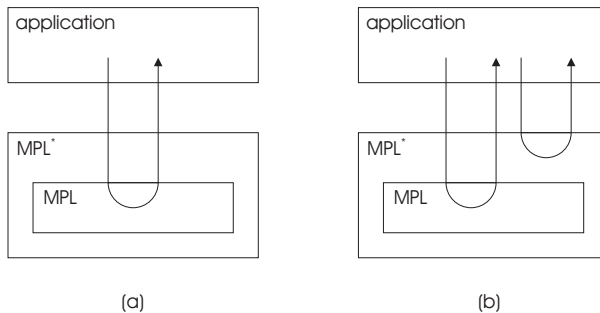


Fig. 3. *MPL** is a layer around the normal *MPL*. During the record phase *MPL** forwards all *MPL*-calls to the *MPL* (part (a)). During the replay phase, *MPL** does forward some calls to the *MPL*, while others call (e.g. the failed *MPI_Test* calls) simply return FALSE (part (b)).

4 Evaluation

The solutions that were proposed in this paper have been implemented in ATHAPASCAN-0 [4,1,2]. ATHAPASCAN-0 is a multi-threaded, portable, runtime system for parallel programs based on threads and message passing. The aims of ATHAPASCAN-0 and similar programming models are to ease the parallel programming of irregular applications, to mask communication and I/O latencies without elaborate efforts from the programmer, to offer a unified framework for exploiting both shared-memory parallelism and message passing [6].

The ATHAPASCAN-0 system is targeted towards hardware systems composed of a network of shared memory multiprocessor nodes. It offers two levels of parallelism: *inter* node and *inner* node. Inter node parallelism is based on a dynamically evolving network of threads, communicating by means of MPI [12]. The inner node parallelism is based on a fixed number of POSIX [13] compliant threads communicating using shared memory. Each ATHAPASCAN-0 node is implemented by a separate MPI process.

Implementing record/replay for ATHAPASCAN-0 requires record/replay for the nondeterministic features of both the thread-library, and the MPI-library. An implementation of record/replay for a *POSIX* compliant thread library has been

described extensively elsewhere [15]. Here, we only focus on the record/replay of the MPI-library. We have inserted our instrumented MPI-library between the ATHAPASCAN-0-library, and the MPI-library. This allowed us to immediately replay ATHAPASCAN-0 program without additional coding.

The execution replay system was tested on several toy ATHAPASCAN-0 programs featuring all possible cases of nondeterminism of the programming model. It was also tested on the available programming examples of the ATHAPASCAN-0 distribution, and it was used to debug the ATHAPASCAN-1 system [3], which represents a large ATHAPASCAN-0 program.

Preliminary test results show that the time overheads during both the record and replay executions remain acceptable while the sizes of the trace files are fairly small. The table below shows execution times (wall time in seconds) for three toy test (not particularly efficient) programs, measured with a confidence range of 95 %. Mandelbrot computes a Mandelbrot set in parallel; queens(12) computes all solutions to a size 12 queens problem while scalprod computes the scalar product of two vectors of 100,000 floating point numbers each. The experiments were performed on two PCs running the Linux operating system and connected by a 100 MBit Ethernet connection.

program	normal	record	replay	logsize (bytes)
mandelbrot	1.858+-0.003	1.914+-0.037	1.864+-0.002	3912
queens(12)	6.70 +-0.15	6.58 +-0.05	6.86 +-0.17	1710
scalprod	4.38 +-0.04	4.44 +-0.07	4.66 +-0.03	874

Surprisingly, the slowdown of the replayed executions remains extremely low. Such a phenomenon stems probably from the behavior of the daemon thread used by ATHAPASCAN-0. When activated, a large part of its activity is probing communications and testing for the completion of non-blocking MPI requests. During replayed executions, most of the calls to *MPL_test* or *MPL_probe* are replaced by simple integer comparisons in the stubs.

5 Conclusion

This article describes a technique to efficiently record and replay the nondeterministic features of message passing libraries, in particular the promiscuous receive and the nonblocking test operations. We have tested out library on the ATHAPASCAN-0 parallel programming systems that is built on top of MPI. These tests have shown that our solution is highly efficient both in time overhead and size of the trace files.

Acknowledgment

This work was partially sponsored by the “Programme d’Actions Intégrés franco-belge Tournesol No. 98114”. Michiel Ronsse is supported by the GOA-project 12050895 of the Universiteit Gent. Koen De Bosschere is research associate with the Fund for Scientific Research — Flanders.

References

1. J. Briat, I. Ginzburg, and M. Pasin. *Athapascan-0 Reference and User Manuals*. LMC-IMAG, B.P. 53, F-38041 Grenoble Cedex 9, March 1998. <http://www.apache.imag.fr/software/ath0/>.
2. J. Briat, I. Ginzburg, M. Pasin, and B. Plateau. Athapascan runtime : Efficiency for irregular problems. In *Proceedings of the Europar'97 Conference*, pages 590–599, Passau, Germany, Aug 1997. Springer Verlag.
3. Gerson G. H. Cavalheiro, François Galilée, and Jean-Louis Roch. Athapascan-1: Parallel Programming with Asynchronous Tasks. In *Proceedings of the Yale Multithreaded Programming Workshop*, Yale, USA, june 1998. <http://www.apache.imag.fr/gersonc/publications/yale98.ps.gz>.
4. J. Chassin de Kergommeaux, M. Ronsee, and K. De Bosschere. Efficient execution replay for athapascan-0 parallel programs. Research Report 3635, INRIA, March 1999. <http://www.inria.fr/RRRT/publications-fra.html>.
5. A. Fagot and J. Chassin de Kergommeaux. Formal and experimental validation of a low-overhead execution replay mechanism. In *Proceedings of Euro-Par'95*, Stockholm, Sweden, August 1995. Springer-Verlag, LNCS.
6. I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37(1):70–82, 1996.
7. M. Hurfin, N. Plouzeau, and M. Raynal. EREBUS A debugger for asynchronous distributed computing systems. In *Proceedings of the 3rd IEEE Workshop on Future Trends in Distributed Computing Systems*, Taiwan, April 1992.
8. H. Jamrozik. *Aide à la Mise au Point des Applications Parallèles et Réparties à base d'Objets Persistants*. PhD thesis, Université Joseph Fourier, Grenoble, May 1993.
9. D. Kranzlmüller and J. Volkert. Debugging point-to-point communication in mpi and pvm. In *Proc. EUROPVM/MPI 98 Intl. Conference*, pages 265–272, September 1998.
10. Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.
11. E. Leu, A. Schiper, and A. Zramdini. Execution Replay on Distributed Memory Architectures. In *Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing*, pages 106–112, Dallas, USA, December 1990.
12. Message Passing Interface Forum, University of Tennessee, Knoxville, Tennessee. *MPI: A Message-Passing Standard*, May 1994.
13. Frank Mueller. A library implementation of POSIX threads under UNIX. In *Proc. of the Winter USENIX Conference*, pages 29–41, San Diego, CA, January 1993.
14. R.H.B. Netzer and B.P. Miller. Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. In *Proceedings of Supercomputing '92*, Minneapolis, Minnesota, November 1992. Institute of Electrical Engineers Computer Society Press.
15. M. Ronsee and L. Levrouw. An experimental evaluation of a replay method for shared memory programs. In E. D'Hollander, G.R. Joubert, F.J. Peters, D. Trystram, K. De Bosschere, and J. Van Campenhout, editors, *Parallel Computing: State-of-the-Art and Perspectives*, pages 399–406. North-Holland, Gent, 1996.

Comparison of PVM and MPI on SGI multiprocessors in a High Bandwidth Multimedia Application

Rade Kutil and Andreas Uhl

RIST++ & Department of Scientific Computing
University of Salzburg, AUSTRIA

e-mail: {rkutil,uhl}@cosy.sbg.ac.at

Abstract. In this work the well known wavelet/subband decomposition algorithm (3-D case) widely used in picture and video coding is parallelized using PVM and MPI as well as with data parallel language extensions. The approaches try to take advantage of the possibilities the respective programming interfaces offer. Experimental results are conducted on an SGI POWERChallenge GR and an SGI Origin 2000. These results show a good comparison of the programming approaches as well as the programming interfaces in a practical environment.

1 Introduction

In recent years there has been a tremendous increase in the demand for digital imagery. Applications include consumer electronics (Kodak's Photo-CD, HDTV, SHDTV, and Sega's CD-ROM video game), medical imaging (digital radiography), video-conferencing and scientific visualization, which all need data compression in order to cope with high memory requirements. Unfortunately many compression techniques demand execution times that are not possible using a single serial microprocessor [13], which leads to the use of general purpose high performance computers for such tasks (beside the use of DSP chips or application specific VLSI designs). In the context of MPEG-1,2 and H.261 several papers have been published describing real-time video coding on such architectures [1, 3].

Image and video coding methods that use wavelet transforms have been successful in providing high rates of compression while maintaining good image quality. Rate-distortion efficient 3-D algorithms exist which are able to capture temporal redundancies (see e.g. [10, 5, 15] for 3-D wavelet/subband coding). Unfortunately these 3-D algorithms often show prohibitive computational and memory demands (especially for real-time applications).

As a first step for an efficient parallel 3-D wavelet video coding algorithm the 3-D wavelet decomposition has to be carried out (followed by subsequent quantization and coding of the transform coefficients). In this work the decomposition step is considered to be the most time consuming and therefore selected for performance comparison.

A significant amount of work has been already done on parallel wavelet transform algorithms for all sorts of high performance computers. We find various kinds of suggestions for 1-D and 2-D algorithms on MIMD computers (see e.g. [16, 14, 12, 6, 4, 7] for decomposition only and [9, 2] for algorithms in connection with image compression schemes). On the other hand, few work has been done (except [11]) focusing especially on 3-D wavelet decomposition and corresponding 3-D wavelet based video compression schemes which are very memory consuming.

Therefore it is interesting to see which programming approach is able to cope with large memory and transmission sizes better in an environment of a practical application.

2 3-D Wavelet Decomposition

The fast wavelet transform can be efficiently implemented by a pair of appropriately designed Quadrature Mirror Filters (QMF). A 1-D wavelet transform of a signal S is performed by convolving S with both QMF's and downsampling by 2. This operation decomposes the original signal into two frequency-bands (called subbands), which are often denoted coarse scale approximation and detail signal. Then the same procedure is applied recursively to the coarse scale approximations several times (see Figure 1 (a)).

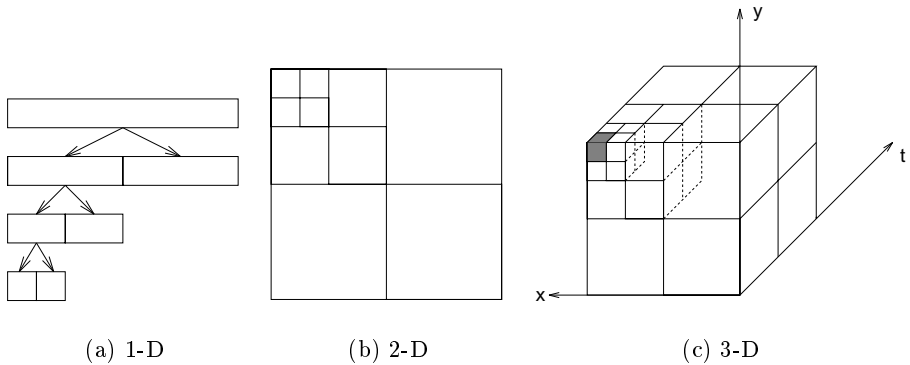


Fig. 1. pyramidal wavelet decomposition.

By analogy to the 2-D case the 3-D wavelet decomposition is computed by applying three separate 1-D transforms along the coordinate axes of the video data. As it is the case for 2-D decompositions, it does not matter in which order the filtering is performed (e.g. a 2-D filtering frame by frame with subsequent temporal filtering, three 1-D filterings along y , t , and x axes, e.t.c.). After one decomposition step we result in 8 frequency subbands out of which only the

approximation data (the gray cube in Figure 1 (c)) is processed further in the next decomposition step. This means that the data on which computations are performed are reduced to $\frac{1}{8}$ in each decomposition step.

Sequential 3-D Wavelet Decomposition - Pseudocode

```

for level=1...max_level {
    for t=1..t_max {
        for x=1...x_max {
            for y=1...y_max {
                convolve S[x,y,t] with G and H } }
        for y=1...y_max {
            for x=1...x_max {
                convolve S[x,y,t] with G and H } } }
        for x=1...x_max {
            for y=1...y_max {
                for t=1..t_max {
                    convolve S[x,y,t] with G and H } } } }
    }
}

```

3 Parallelization of 3-D Wavelet Decomposition

3.1 Message Passing

When computing the 3-D decomposition in parallel one has to decompose the data and distribute it among the processor elements (PE) somehow. In previous works data is distributed along the time-axis [8] or along spatial axes [11] into parallel-epipeds (see Figure 2). In this work both variants have been used.

In the literature two approaches for the boundary problems have been discussed and compared [16,14]. During the *data swapping* method (also known as *non-redundant data calculation*) each processor calculates only its own data and exchanges these results with the appropriate neighbour processors in order to get the necessary border data for the next decomposition level. Employing *redundant data calculation* each PE computes also necessary redundant border data in order to avoid additional communication to obtain this data.

In this work we employ a combined approach. One can choose, how many of the parallel steps should be processed using redundant data. The succeeding steps then have to swap border data. Figure 2 shows the stages of the algorithm.

- (1) The video data is distributed uniformly among the PE (including redundant data as necessary).
- (2) The PE exchange the border data (light gray) if necessary for the next decomposition level with their corresponding neighbours.
- (3) The 3-D decomposition is performed on the local data on each PE.
- (4) All subbands but the approximation subband (dark gray) are collected (there is no more work to do on these data).
- (5) Repeat steps 2 - 4 until the desired decomposition depth is reached.

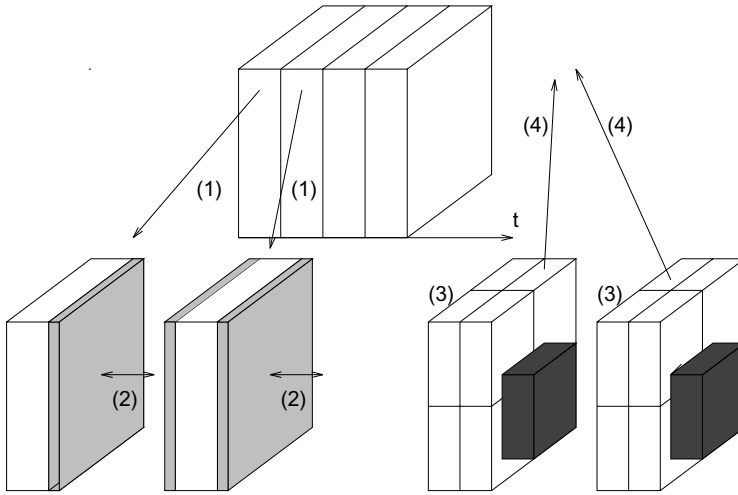


Fig. 2. Message passing with data swapping.

There are several improvements to this scheme:

- One can easily see, that with higher numbers of processors the splitting of the video data produces slices which are so thin, that the border data (that is of the size of the filter length) gets bigger than the slices, which would be very inefficient. One can therefore split data not only in time domain but in spatial domains too, hence distributing subcuboids of data among the PE. This also can reduce the amount of border data that has to be transmitted in each step as a short calculation in section 4 shows.
- It is a fact, that the processes are executed very asynchronously. That is because the last node PE has to wait very long in the start-up phase for its part of the video data, so that it starts calculation at a later time. Especially when splitting the data in several dimensions as supposed above the exchange of border data can lead to extensive communication that resynchronizes the processes, whereby time is lost. As a solution data can be sent in several parts in the beginning. The node PE can then perform the according part of the first decomposition step and then wait for the next part to be received.

3.2 Data Parallel Implementation

Data parallel programming on a shared memory architecture is easily achieved by transforming a sequential algorithm into a parallel one by simply identifying areas which are suitable to be run in parallel i.e. where different iterations access different data. Subsequently, local and shared variables need to be declared and parallel compiler directives are inserted. This was done with IRIS PowerC using pragma statements.

3.3 Investigation of the abilities of PVM and MPI

It is of great advantage, if processes can continue calculation even if sent data is not yet received by other PEs. The only way to do this with PVM is to provide large communication buffers. On shared memory implementations of PVM it is often possible to specify a shared memory buffer size. This size has to be increased to get good performance and even to avoid deadlocks. Also the transmission of a large data block has to be divided in smaller pieces.

With MPI, the standard `scatter` and `gather` calls provided by MPI could not be used, because of too complex data structures (3-D array). It is not possible to specify one data type that can be used for all subcuboids of the 3-D array, because they do not have equal sizes. Therefore specifying different data types for each data block to be sent/received would have to be necessary, which is not possible even in MPI2.

As to avoid unnecessary copying of data we use non blocking sends and receives instead, and start waiting for their termination at the latest possible time (i.e. after performing calculations which do not affect the data that is still in access). Therewith the hope for improved performance arises.

4 Experimental Results

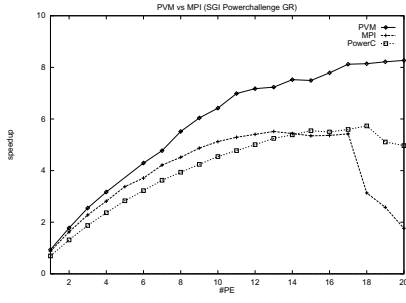
Experiments were conducted on an SGI POWERChallenge GR (at RIST++, Salzburg Univ.) with 20 MIPS R10000 processors and 2.5 GB shared memory and an SGI Origin 2000 (at Linz Univ.) with 30 MIPS R10000 processors. On both IRIX64 release 6.5 is installed. The size of the video data is 256×256 pixels in the spatial domain, combined to a 3-D data block consisting of 512 frames. QMF filters with 8 coefficients were used which is a common size in picture coding applications. The message passing libraries used are SGI-MPI 3.1.1.0 and a special shared memory variant of PVM for SGI systems (SGI PVM 3.3.10).

Figure 3 shows the results of test runs on the two computers mentioned above. The left subfigure additionally shows the speedups for the data parallel implementation with IRIS PowerC.

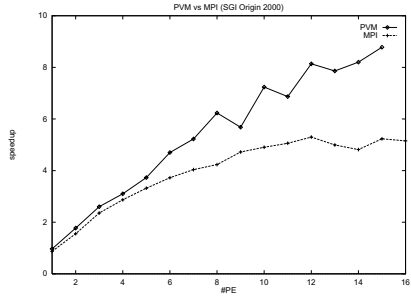
The relatively poor performance of MPI could not be overcome. A reason for the performance breakdown with high numbers of PEs could not be found. The results for PowerC show, that the data parallel paradigm is not able to reach the performance of message passing.

Within Figure 4 two runs with the same parameters with eight PEs are shown, which should illustrate, how the worse performance of MPI is achieved. The lowest horizontal line symbolizes the progression of the master PE in time, the other represent the node PEs. The fat black parts of these time lines represent calculation phases. Crossed lines and gray parts symbolize the transmission of data from one PE to the other, where the gray parts result from the time that is consumed between the beginning and end of a send or a receive. Time is measured in seconds.

One can see, that there is no algorithmic problem, but each communication step is simply slower than in PVM. Calculation times are not affected.

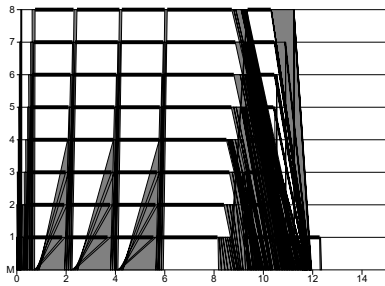


(a) on SGI Powerchallenge GR

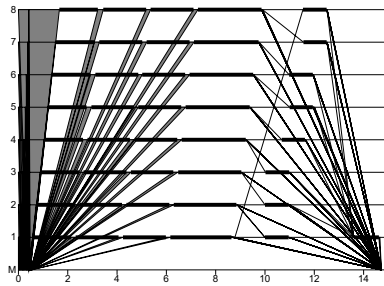


(b) on SGI Origin 2000

Fig. 3. Comparison: PVM vs MPI

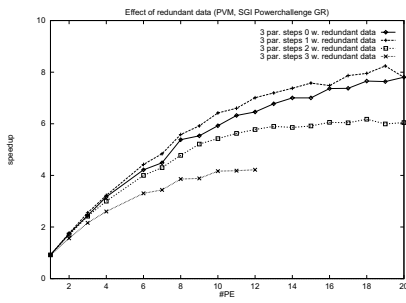


(a) PVM

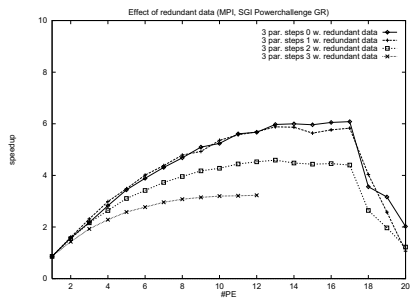


(b) MPI

Fig. 4. Execution scheme of a wavelet decomposition (on SGI PowerChallenge GR)



(a) PVM



(b) MPI

Fig. 5. Redundant data levels

A big question in the topic of parallel wavelet decomposition is, whether redundant data should be used or data swapping as explained in section 3.1. Figure 5 shows, that this question has to be answered differently for PVM and MPI. Where PVM is able to achieve a performance gain by sending redundant data for the first decomposition step, MPI even slows down.

Apart from that, the general behavior is the same. Too much redundant data raises the communication data sizes in the start-up phase as well as calculation times.

The difference in splitting the data along several axes instead of just one can be demonstrated by the following calculation. As we use data of the size of $512 \times 256 \times 256$ pixels, border data normally has a size of $6 \times 256 \times 256$ pixels. 6 is the overhead for a filter length of 8. If we split the data 4 times in time domain and 2 times in spatial domain, one block has the size of $128 \times 128 \times 128$. Border data therefore is $(128 + 6)^3 - 128^3$, which is about 30% less. But on the other hand, the communication structure gets more complex.

Now, if performance is better in this case, the data size should be the bottle neck. If there are performance differences between PVM and MPI, one could detect characteristics of each of them. But as Figure 6 does not show any significant differences neither between types of data splitting nor between PVM and MPI, no statement can be made. But one can expect different results for systems with lower bandwidth (workstation clusters).

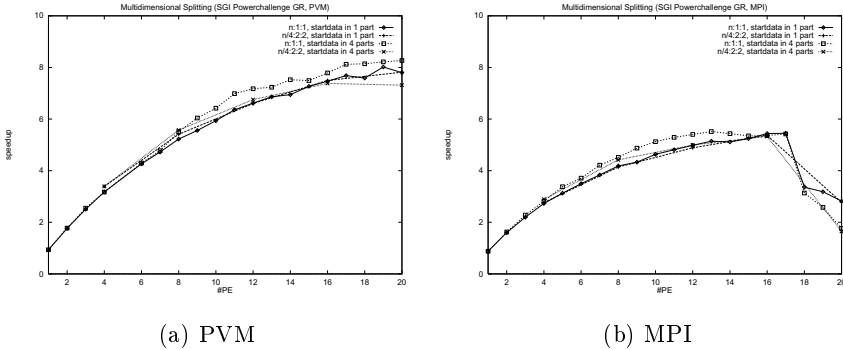


Fig. 6. Parallelizing along several axes

Acknowledgements

The author was partially supported by the Austrian Science Fund FWF, project no. P11045-ÖMA.

References

1. S.M. Akramullah, I. Ahmad, and M.L. Liou. A data-parallel approach for real-time MPEG-2 video encoding. *Journal of Parallel and Distributed Computing*, 30:129–146, 1995.
2. C.D. Creusere. Image coding using parallel implementations of the embedded zerotree wavelet algorithm. In B. Vasudev, S. Frans, and P. Sethuraman, editors, *Digital Video Compression: Algorithms and Technologies 1996*, volume 2668 of *SPIE Proceedings*, pages 82–92, 1996.
3. A.C. Downton. Generalized approach to parallelising image sequence coding algorithms. *IEE Proc.-Vis. Image Signal Processing*, 141(6):438–445, December 1994.
4. J. Fridman and E.S. Manolakos. On the scalability of 2D discrete wavelet transform algorithms. *Multidimensional Systems and Signal Processing*, 8(1–2):185–217, 1997.
5. B.J. Kim and W.A. Pearlman. An embedded wavelet video coder using three-dimensional set partitioning in hierarchical trees (SPHIT). In *Proceedings Data Compression Conference (DCC'97)*, pages 251–259. IEEE Computer Society Press, March 1997.
6. C. Koc, G. Chen, and C. Chui. Complexity analysis of wavelet signal decomposition and reconstruction. *IEEE Trans. on Aerospace and Electronic Systems*, 30(3):910–918, July 1994.
7. D. Krishnaswamy and M. Orchard. Parallel algorithm for the two-dimensional discrete wavelet transform. In *Proceedings of the 1994 International Conference on Parallel Processing*, pages III:47–54, 1994.
8. R. Kutil and A. Uhl. Hardware and software aspects for 3-D wavelet decomposition on shared memory MIMD computers. volume 1557 of *Lecture Notes on Computer Science*, pages 3347–356. Springer-Verlag, 1999.
9. G. Lafruit and J. Cornelius. Parallelization of the 2D fast wavelet transform with a space-filling curve image scan. In A.G. Tescher, editor, *Applications of Digital Image Processing XVIII*, volume 2564 of *SPIE Proceedings*, pages 470–482, 1995.
10. A.S. Lewis and G. Knowles. Video compression using 3D wavelet transforms. *Electronics Letters*, 26(6):396–398, 1990.
11. H. Nicolas, A. Basso, E. Reusens, and M. Schutz. Parallel implementations of image sequence coding algorithms on the CRAY T3D. Technical Report Supercomputing Review 6, EPFL Lausanne, 1994.
12. J.N. Patel, A.A. Khokhar, and L.H. Jamieson. Scalability of 2-D wavelet transform algorithms: analytical and experimental results on coarse-grain parallel computers. In *Proceedings of the 1996 IEEE Workshop on VLSI Signal Processing*, pages 376–385, 1996.
13. K. Shen, G.W. Cook, L.H. Jamieson, and E.J. Delp. An overview of parallel processing approaches to image and video compression. In M. Rabbani, editor, *Image and Video Compression*, volume 2186 of *SPIE Proceedings*, pages 197–208, 1994.
14. S. Sullivan. Vector and parallel implementations of the wavelet transform. Technical report, Center for Supercomputing Research and Development, University of Illinois, Urbana, 1991.
15. D. Taubman and A. Zakhor. Multirate 3-D subband coding of video. *IEEE Transactions on Image Processing*, 5(3):572–588, September 1993.
16. M-L. Woo. Parallel discrete wavelet transform on the Paragon MIMD machine. In R.S. Schreiber et al., editor, *Proceedings of the seventh SIAM conference on parallel processing for scientific computing*, pages 3–8, 1995.

On Line Visualisation or Combining the Standard ORNL PVM with a Vendor PVM Implementation

Janusz Borkowski

Polish-Japanese Institute for Information Technology
Koszykowa 86, 02-008 Warsaw, Poland
janb@pjwstk.waw.pl

Abstract. Vendor implementations of PVM provide optimum performance on multicomputers. However, such implementations may lack interoperability. Developing of a real time display for a LGA simulation on Hitachi SR2201 parallel computer required a connection between the calculating part running on SR2201 and GUI running on a graphics workstation. A native PVM could not provide it. We proposed to use ORNL PVM to facilitate the connection and at the same time to use the native PVM for effective internal communication. Performance evaluation of this approach and practical results are presented.

1. Introduction

Parallel applications are often run in batch mode. However, sometimes interactivity is desirable or even necessary. Interactivity can mean just a simple window with controls suitable to set parameters, start and stop, or it can mean a sophisticated visualisation environment with life display. As for the latter, it is not unusual to divide a parallel application into two parts:

- Calculating part or back-end, which performs number crunching, it runs on a parallel supercomputer.
- User interface or front-end, responsible for a visual presentation and an interaction with the user. This part runs on a workstation where the visualisation environment performs best.

Surely both parts must be somehow connected. GUI (Graphics User Interface) sends parameters and commands to the back-end and receives results to be shown.

Similar task —connecting a parallel supercomputer with other machines —must be done in a general situation when one wants to use combined computing power.

PVM [10] can usually do such jobs excellently. Unfortunately some parallel supercomputers are supplied with a vendor PVM implementation, which performs very well, but which is unable to incorporate any other computer in the virtual machine. It means that only CPUs belonging to a parallel computer can be used by a vendor-PVM application. This paper takes such a case into account and discusses possibilities of extending vendor-PVM functionality to overcome the above mentioned problem. A solution for Hitachi SR2201 parallel computer is shown.

Paragraph 2 informs about the previous work of the author, from which this paper stems. Paragraph 3 presents the test application —LGA. SR2201 characteristics and its interprocess communication capabilities are presented in p. 4. Our solution for external PVM communication —double PVM —is described and evaluated in p. 5. Paragraph 6 shows the effects of employing double PVM in the LGA simulation program. Finally p.7 summarises the results of this work.

2. Motivation for this Work

In the previous paper of the author [1] a construction of a life display connected to a parallel application was presented. Various benefits taken from employing a GUI instead of batch processing were discussed, too. Java language was used to implement the GUI. Performance of Java graphics and Java-to-PVM bindings called jPVM [4] were shown. Although a working application capable of displaying moving pictures has been constructed, some shortcomings remained unsolved.

Experiments were done on a workstation cluster. A Hitachi SR2201 supercomputer [6] was also available but there were difficulties in putting it to work. SR2201 does not support Java language and Hitachi PVM implementation does not provide functions concerning connecting to other computers. So, when the computational part of an application was to be run on SR2201, then GUI was neither able to run on SR2201 nor able to connect to it from any other computer.

Java is available for a large number of platforms and it is free. However, implementing a comprehensive GUI with advanced visualisation capabilities is not a simple task. On the other hand proven visualisation software packages exist, there is a possibility to use them for state-of-the-art GUI and data display.

This work deals with both problems mentioned above.

3. LGA

We developed a sample parallel application to be used in our work as a testbed. The application is a Lattice Gas Automata simulation program.

Lattice Gas Automata was proposed in 1973 by Hardy, de Pazzis and Pomeau [7] as a simple method to study ergodicity related problems. Their model, called HPP, is based on a square lattice whose nodes can be occupied by “fluid particles”. HPP model presents only limited application because square lattice leads to anisotropic Navier-Stokes equations. More accurate models were developed eg. FHP based on hexagonal lattice (Frisch, Hasslacher, Pomeau [8]) or FCHC used for 3 dimensional simulations. Those models are used for fluid flow modelling and study of diffusive phenomena.

We used HPP model in this work due to its simplicity, and also because we were interested mainly in parallel application design problems. More information on LGA and literature can be found at [9].

4. PVM Modification Possibilities

It is quite common, that a supercomputer supports a proprietary version of a communication library. Such a library works fine within a particular machine, but cannot be used for external communication. Some work has been done already to alleviate such a problem. In [3, 4] a connection between separate MPI environments is described. Computers use MPI for internal communication and PVM as a bridge between them. In [2] an extended MPI implementation is presented. StaMPI, as it is called, can facilitate communication between separate supercomputers of different types.

We had an intention to provide a kind of similar solution for PVM-only communication.

4.1. SR2201, Its Communication Libraries and OS

Hitachi SR2201 [6] is a MIMD type machine. Each CPU has its private memory. Internal network (2 or 3 staged crossbar) is capable of transmitting 300MB/s from one CPU to another. A low level library, so called "Remote-DMA" can be used to fully utilise this hardware. Other native communication libraries on SR2201 (PVM, MPI, Express) use the Remote-DMA layer to facilitate actual data movement and synchronisation. PVM reaches bandwidth 70MB/s.

The operating system, called HI-UX/MPP, is a microkernel based Unix type OS. The important feature is that on an usual CPU, called PE, only a microkernel resides. Unix system calls are performed by a Unix server process, running on a dedicated CPU(s), called IOU(s). Unix calls must be forwarded to an IOU, carried out there and then a result can be sent back. In our belief such a design (or/and its implementation) causes some shortcomings in UNIX performance. We will deal with this issue later.

4.2. Communication Performance Based on HI-UX/MPP Socket Interface

The standard PVM uses sockets for data exchange between tasks (processes). A port of the standard PVM to SR2201 was feasible, but first we wanted to estimate the performance of such a solution. We measured bandwidth of a process to process communication using sockets directly. AF_UNIX socket family was used to get highest possible rates (upper bound) for socket communication. Figure (fig. 1.) summarises the results. AF_INET socket performance is given for comparison.

A comparison between values obtained from fig. 2 and the SR2201 characteristics shows, that the socket interface on SR2201 performs about 10 times slower than a native PVM implementation. Unfortunately, tests revealed that a straightforward port of ORNL PVM can achieve only about 250kB/s PE-to-PE bandwidth. We try to explain it below. Anyway, a port of the standard PVM to SR2201 without deep modification does not make much sense. For internal message transmission RDMA should be used. That is what Hitachi PVM does, but we do not have any access to the sources to be able to modify it.

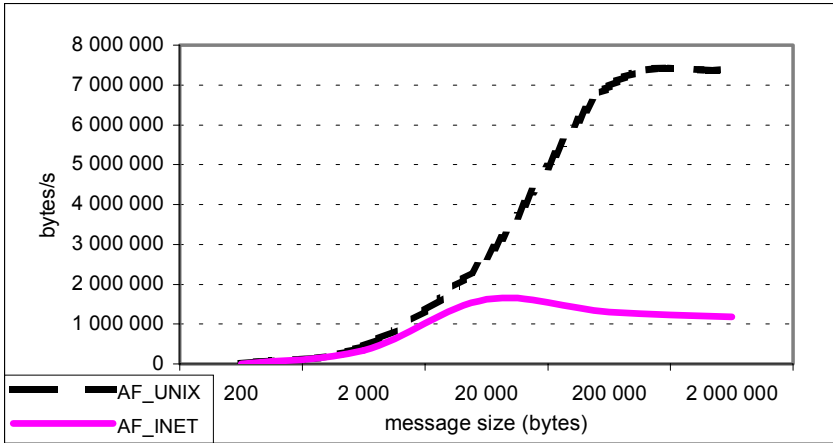


Fig. 1. Sustained bandwidth in SR2201 PE-to-PE communication using TCP sockets. Pingpong test.

Socket interface is necessary anyway for external communication. Interoperable PVM has to use it. We compiled the standard PVM library on SR2201 to check how the external communication performs. The results are surprisingly good (comparing to PE-to-PE results): 2 400 000 B/s using Ethernet 100BaseT was achieved (see also fig. 3.). We can explain it like that: in case of an internal message exchange the Unix server has to work for both communicating processes. It serializes data movement operations and it may easily become overloaded. In case of external message exchange, providing that *PvmRouteDirect* option is employed, only one process on SR2201 requests Unix services. Without *PvmRouteDirect* the results are 3 times (for short messages) to 12 times (for long messages) worse.

5. Double PVM

From the observations made so far it can be stated that message exchange inside SR2201 must rely on RDMA to get reasonable performance. Socket interface must be used by an interoperable PVM for external communication. Hitachi PVM fulfils the first condition, ORNL PVM is compliant with the second one. Let us use both PVM versions in concert.

5.1. Proposed Idea and Its Implementation

Our idea is to have one process (or a few if necessary) running on SR2201 connected to both Hitachi PVM and ORNL PVM. Such a process —called a router—will pass messages between tasks running on other computers and tasks running within SR2201

(fig. 2). To be able to do this, both PVM libraries must be linked to such a process and name conflicts have to be resolved. Our approach was to change all exported names in the standard PVM library. It was done using *sed* editor with a few command scripts. All *pvm* prefixes were changed to *pvmornl* or *ornl* suffix was added.

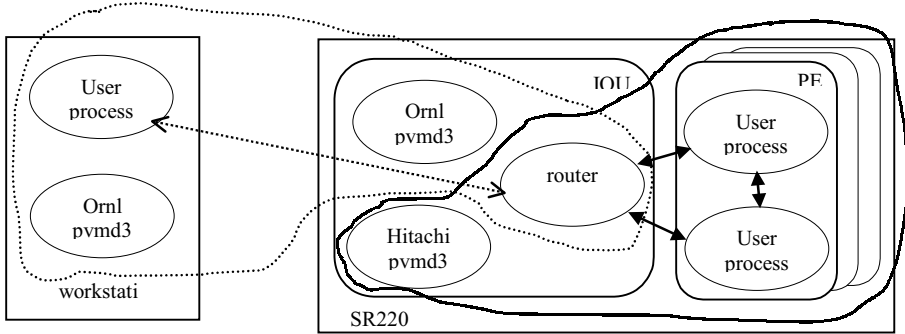


Fig. 2. Message exchange paths for “double PVM”. Solid arrows indicate *pvm_messages* (native implementation), dotted arrows indicate *pvmornl_messages* (ORNL implementation).

5.2. Performance of “Double PVM” Solution

We measured the bandwidth of task-to-task communication using a pingpong test. One task was run on a workstation connected by 100BaseT Ethernet to the SR2201. The second task worked as a router, it was run on an IOU on SR2201. Finally the third task was placed on a PE on SR2201. The results are given in figure 3. It can be seen, that the routing does not introduce much overhead (coefficient 0.74 to 0.9). Again *PvmRouteDirect* option improved results very significantly: 4 (for short messages) to 13 (for long messages) times.

Figure 4 compares the bandwidth of PE-to-workstation communication using “double PVM” with direct TCP socket connection. Double PVM performs significantly worse for short messages (only 30% of the socket results), which means that a latency is rather high (table 1). Fortunately transmitting simulation results, which is our point of interest, requires much bigger packets. For 2MB message size 92% of a socket connection throughput was observed.

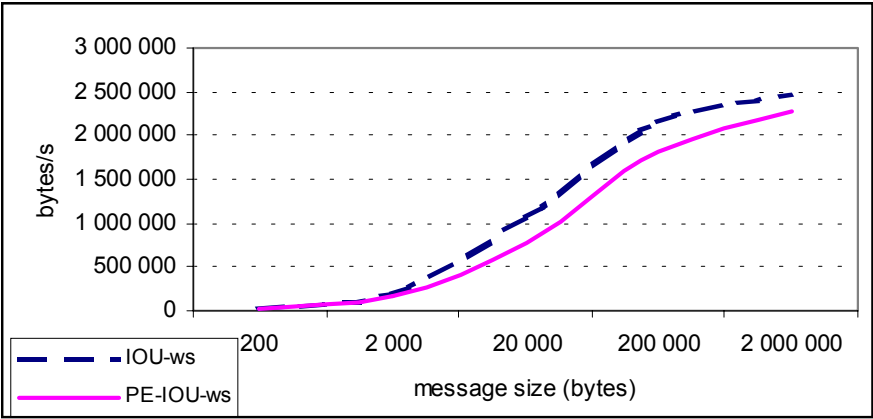


Fig. 3. Sustained bandwidth in SR2201-to-workstation PVM communication, pingpong test. IOU-ws connection (upper line) uses ORNL PVM only, PE-IOU-ws connection (lower line) uses Hitachi PVM for PE-IOU part and ORNL PVM on Ethernet 100BaseT for IOU-ws part. PvmDataInPlace used for both PVMs, PvmRouteDirect used for ORNL PVM.

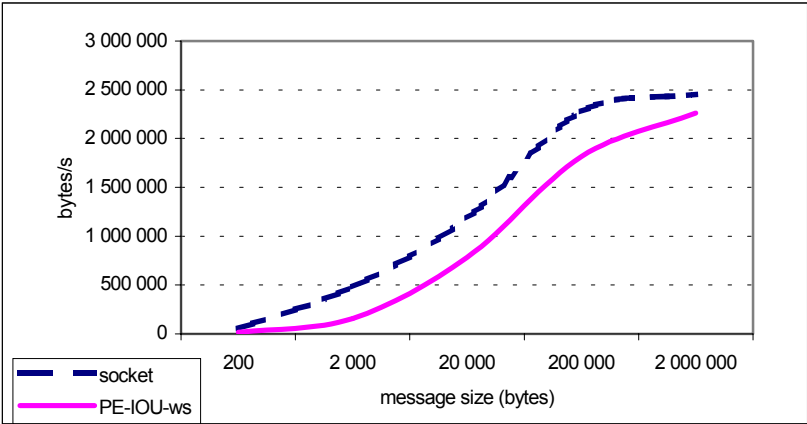


Fig. 4. Sustained bandwidth between SR2201 PE and a workstation connected by Ethernet 100BaseT, pingpong test: direct TCP socket connection vs. “double PVM” connection. PVM parameters for PE-IOU-ws are same as in figure 3.

Table 1. Average latency times measured as half of a round trip time of a single byte message. PVM settings same as for fig. 3.

test description	μsec
Hitachi PVM, PE to PE	213
ORNL PVM, workstation to IOU	11300
double PVM, workstation to IOU to PE	16 500
socket connection, workstation to PE	2 050

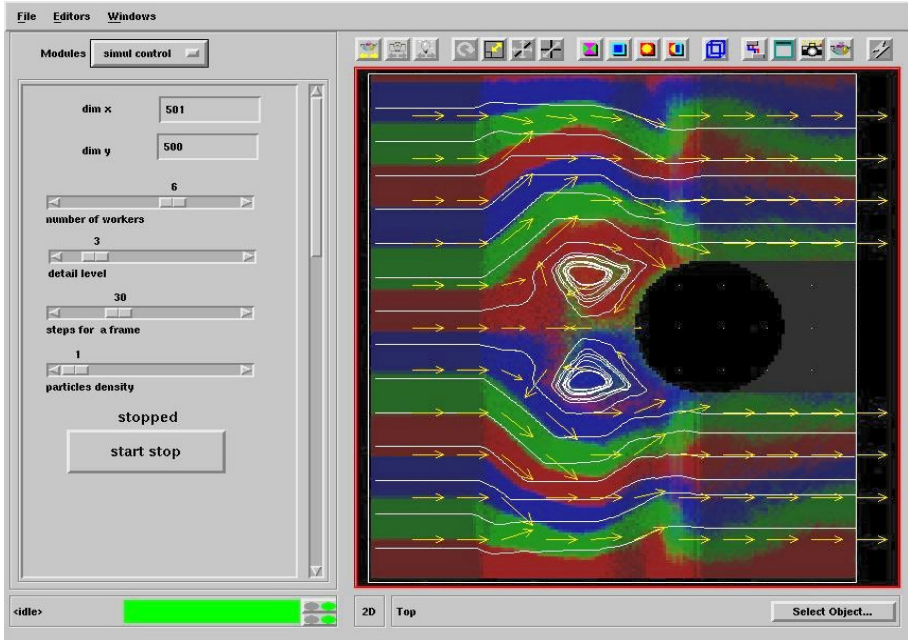


Fig. 5. AVS version of GUI. Three kinds of visualization techniques applied: colored stripes (poorly visible in black-and-white printout), vector samples and streamlines.

6. Practical Results

Our LGA simulation program [1] was modified to fit in with the new concept. Additional process —a router—has been added. The router process has to be started first. It registers its TID in the PVM database and wait for a GUI to connect. Any GUI from any place (within PVM configuration) can make the connection and start a simulation. Worker processes are created then in a number specified by the user. We developed two version of GUI. The first version is written in Java. It is a modifieds GUI described in [1]. The second version is based on AVS [11] and is much more flexible. The key point here is a custom made AVS module. This module provides an user interface and it is responsible for data transfer—it contains PVM calls. The module can connect to a router process, then user commands can be send and simulation data can be received. An arbitrary AVS network can process these data as necessary, see example on fig 5.

7. Conclusions

The native implementation of PVM on Hitachi SR2201 is not fully compliant with the standard PVM: it cannot connect to other computers. We presented a method to overcome this drawback. Native PVM can be used for effective communication within SR2201, the standard PVM facilitates external message exchange. A dedicated process(es) can be employed to relay messages between the different PVM implementations. Achieved bandwidth between a process running on SR2201 and a process running on an other computer varies from 0.3 (200 bytes message) to 0.92 (2MB message) of the bandwidth provided by a direct socket connection. An example application was developed making use of the idea. Part of the application performed calculations on the SR2201, another part, running on a workstation, provided GUI displaying the results of the calculations in real time. It was verified, that “double PVM” concept can be implemented without much effort and it can be successfully used in practice.

References

1. Janusz Borkowski.
2. T. Kimura and H. Takemiya: Local Area Metacomputing for Multidisciplinary Problems: A Case Study for Fluid/Structure Coupled Simulation, in Proceedings of International Conference of Supercomputing ICS 98 Melbourne, Australia, ACM 1998 0-89791-998-X pp 149-156.
3. Graham Fagg and Jack Dongarra, PVMPI: An Integration of PVM and MPI Systems, *Calculateurs Paralleles*, Volume 8, Number 2, 1996, pp 151-166, Hermes, ISSN: 1260-3198.
4. T. Beisel, E. Gabriel and M. Resch, “An Extension of MPI for Distributed Computing on MPPs” in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, LNCS, Springer 1997, pp. 75-83.
5. <http://www.isye.gatech.edu/chmsr/jPVM/> jPVM: A Native Methods Interface to PVM for the Java Platform.
6. H. Fujii, Y. Yasuda, H. Akashi, Y. Inagami, M. Koga, O. Ishihara, M. Kashiya, H. Wada and T. Sumitomo, “Architecture and performance of the Hitachi SR2201 Massively Parallel Processor System” *Proceedings of the 11th International Parallel Processing Symposium*, Geneva 1997.
7. J. Hardy, Y. Polmeau & O. de Pazzis, “Time evolution of two-dimensional model system. I. Invariant states and time correlation functions.” *J.Math. Phys* 14 (1973) pp. 1746-1759.
8. U. Frisch, B. Hasslacher & Y. Pomeau, “Lattice-gas automata for Navier-Stokes equation” *Phys. Rev. Lett.* 56 (1986) pp. 1505-1508.
9. http://poseidon.ulb.ac.be/simulations/intro_en.html Introduction to lattice gas automata.
10. V.S. Sunderam “PVM: A Framework for Parallel Distributed Computing”, *Journal of Concurrency: Practice and Experience* pp. 315-339, Dec 1990.
11. <http://www.avs.com> Home Page of Advance Visual Systems.

Native versus Java Message Passing

Nenad Stankovic and Kang Zhang

Department of Computing, Macquarie University
NSW 2109, Australia
nstankov@comp.mq.edu.au, kang@ics.mq.edu.au

Abstract. As an objected-oriented programming language and as a platform independent environment Java has been attracting much attention. However, the trade-off between portability and performance hasn't spared Java. The initial performance of Java programs has been poor, due to the interpretive nature of the environment. In this paper we present the communication performance results for three different types of message-passing programs: native, Java and native communications, and pure Java. Despite concerns about performance and numerical issues, we believe the obtained results confirm that high-performance parallel computing in Java is possible, as technology matures.

1 Introduction

When programming for parallel processing, the message-passing model, although less ambitious, has proved to be more popular than the shared memory. The main advantages are generally better performance, flexibility, and integrability into the existing programming tools and practices. Message passing libraries like the Message Passing Interface (MPI) [3] and the Parallel Virtual Machine (PVM) [2] provide a common programming interface and a portable program source code across many computer architectures that make up a network or cluster. The library interface is standardized, but multiple architecture and topology specific and performance-tuned implementations of the same library are available. Different implementations employ different approaches, and thus, the performance of each implementation may deviate. When programming in languages like C and FORTRAN it is nevertheless necessary to provide a correct executable for each target architecture.

In 1995 Sun Microsystems introduced a form of machine independent binaries that allow executables to travel over the Internet and execute on any host machine containing the needed interface, referred to as a Java Virtual Machine (JVM). This new language entitled Java quickly found a home in the entangled Internet Web community, with programmers writing small applications called applets. The fundamental trade-off between portability and performance is well known to programmers of scientific applications for parallel processing. With an interpretive language like Java we are looking at an order of magnitude or more slower execution speed. Making Java programs run faster is a challenging task and active area of research. Besides the interpreter, the other approaches to running Java are a hardware implementation, a Hot Spot compiler and a Just-In-Time (JIT) compiler. The most

common solution to performance problems is by the use of a JIT that translates Java bytecodes to native instructions, at runtime. The compilation causes some initial slowdown in program execution speed, but significant performance improvements have been observed and reported [6]. The Hot Spot technology is even more promising since it can further optimize the code by performing runtime analysis of the frequency of the code segments execution. On the other hand, the Java bytecode language is emerging as a software distribution standard, with major software and hardware vendors committed to porting the Java run-time environment to their platforms. Being an object-oriented language, Java also has several built-in mechanisms that allow exploitation of the parallelism that is inherent in scientific computing. Java threads and concurrency constructs that make use of native threads are well suited to shared-memory computers. The performance of a parallel program is also influenced by the speed that data can be exchanged between the running processes. The Java networking package provides communication classes based on sockets and the Remote Method Invocation (RMI) [12] that can be used for message-passing programming, but are regarded rather low-level and their scalability is questionable. The way the Java I/O has been implemented was found to be a major source of overhead in some benchmarks [4]. The ability to access standard native libraries from Java programs through Java Native Interface (JNI) [10] is important not only for performance reasons, but also for reusing the wealth of existing C and FORTRAN code with very little cost when writing new applications in Java.

In this paper we present a comparison between three different combinations in which programs for parallel processing could be found. In Sections 2 and 3, we first briefly describe the libraries under consideration. We have installed and tested two native libraries, a PVM and a MPI implementation, and two Java libraries that provide interfaces to these two native libraries. In Section 4 we describe our own communication library called VComms, which is a pure Java message passing library that follows to a certain extent the MPI standard, but also takes advantage of the features provided by Java. In Section 5 we present and compare the obtained results for all the presented systems, some of them collected in a heterogeneous networking environment. Section 6 concludes the paper.

2 MPI and PVM

Message-passing model has become the paradigm of choice for parallel processing. Although there are many variations, the basic concept is well understood and has been widely exploited for SPMD type of programs on parallel computers and networks of workstations (NOW). MPI has become an emerging standard for implementing SPMD, and since the release of the initial MPI specification [7], several MPI implementations have been made publicly available. The standard is primarily concerned with message-passing issues and performance, and leaves the question of process creation open. The LAM programming environment and development system [8] we have used for our tests provides, nevertheless, a complete set of tools to compile, run and debug parallel programs.

Similar to MPI, PVM is used to transform a network of computers into a metacomputing environment. Although less rich in different modes and

communication primitives, a PVM implementation is a self-contained system that provides utilities to spawn and control processes. A PVM program is implemented as a set of processes that may join and leave the application, thus promoting a dynamic processing environment.

3 JavaMPI and jPVM

For Java to establish itself in scientific programming, the interoperability with the native legacy software through the JNI is very important. This is also important for performance reasons, especially when no method apart from interpreting is used to execute the bytecodes. In principle, the binding of a native library to Java can be accomplished by either dynamically linking the library to the JVM, or linking the library to the object code generated by a stand-alone Java compiler. Complications stem from the fact that Java data formats are different from those in C. Therefore, a native method interface allows C functions to access Java data and perform format conversion if necessary. In binding a native library to Java portability problems may arise. The JNI was not part of the original specification, and incompatible interfaces exist from different vendors.

Nevertheless, in an effort to combine the advantages of the new features offered by Java, and yet do not suffer too much on performance, access to communication libraries like MPI and PVM has been enabled. The JavaMPI [5] binding allows Java programs to use a native MPI library, which was LAM in our case. It comes with the Java-to-C interface generator (JCI) that takes as input a C header file and generates a Java native method declaration for each exported function. Thus generated stubs perform arguments conversion into the form understandable to the corresponding C functions. The JavaMPI consists of more than 120 functions. The jPVM [14] interface is similar to JavaMPI in its approach, but provides interoperability between Java and PVM.

4 VComms

Four key aspects of Java bytecode-based software distribution motivate our interest in Java. First, Java adds security to the distribution of software [13], providing added benefits over more common languages like C and C++. Second, JVM defines an interface for executing Java bytecode programs on a wide variety of architectures, allowing for the development of a truly portable software base. Third, Java contains many features that are vital for the success of any universal language. Last, due to the nature of interpreted languages, Java executables run slower than their compiled counterparts in other languages. We investigate solutions to overcome this problem.

VComms is a communication class (library) that follows the MPI standard, but also takes advantage of the features like function overloading and object serialization [11] available in Java to simplify the implementation and programming. To help with data marshaling, all message-passing methods take serializable objects as arguments, and consequently messages consist of serializable Java objects that are communicated

via sockets. When serializing data, the programmer does not have to define the data type or the array size. Being properties of the data types, that is taken care of by the serialization mechanism itself. These features simplify the signature of the MPI calls in our implementation and reduce the possibility of errors. As in MPI, each message has a tag, a process ID and a group to which the sending or receiving process belongs. These three attributes are combined into a VDataSend object. All calls that send data take another argument that represents the contents. For example, a blocking send call is defined as:

```
Send(VDataSend(tag,toProcID,group),userData);
```

and the matching blocking receive:

```
userData = Recv(VDataRecv(tag,fromProcID,group));
```

The implementation supports blocking and non-blocking point-to-point and collective messaging, together with synchronization capabilities in a raw and trace mode. At the moment, the inter-group communication is enabled only in the point-to-point mode. Synchronous and asynchronous calls can be combined together, at both ends of the communication channel. VComms is a part of our metacomputing environment called Visper [9].

5 Communication Benchmarking

To send a message from one node to another the following steps are required:

- Network address resolution
- Data marshaling
- Data transfer

In message-passing programming, the network address resolution involves the conversion of a logical process identifier with respect to a communication group into an actual IP address and port number. The data marshaling involves the conversion of the data from the local host format into the network format, and vice-versa. The data transfer comprises the establishing of a reliable communication channel. These three steps were included in the benchmarking results presented below.

5.1 The Environment

We have used the following software in our tests:

- LAM6.2b (University of Notre Dame) compiled with gcc2.7.2 on UltraSparc
- PVM 3.3 compiled with gcc2.7.2 on UltraSparc
- JavaMPI and jPVM as downloaded
- jdk 1.1.6 without a JIT from Sun Microsystems for UltraSparc/Solaris
- jdk/jre 1.2 from Sun Microsystems for PC/NT
- HP-UX Java C.01.15.05

The operating systems were Solaris 2.5 on Sparc, NT 4.00.1381 on PC and HP-UX B.10.20. The hardware is defined in Table 1. The network was a 10 Mbps Ethernet. We have used 2 same Sun and HP boxes and 2 PCs.

Table 1 Hardware

Vendor	Architecture	RAM(MB)	CPU(MHz)
Sun	Ultra 2	256	168 * 2
HP	A 9000/780	512	180
Compaq (PC)	Pentium II	64	200
Micron (PC)	Pentium II	64	200

5.2 COMMS1 Benchmark

The purpose of the COMMS1 benchmark [1] is to measure the basic communication properties of a message-passing computer by measuring the end-to-end delay. Also known as the pingpong benchmark, it requires 2 processes, where each process acts as a sender or a receiver interchangeably. The communication times for blocking calls were measured for messages of various lengths, ranging from 1 to 1000000 bytes.

Table 2 LAM and PVM Times (ms)

Bytes sent	LAM	PVM	javaMPI	jPVM	jPVM!d
1	0.358	0.579	1.08	0.95	1.4
100	0.540	0.766	0.78	1.10	1.6
1000	2.043	2.363	2.28	2.65	3.15
10000	9.987	10.41	10.2	10.8	15.9
100000	91.99	95.84	97.1	97.6	140.5
1000000	922.3	966.2	957.7	985.1	1394

Table 2 presents the results collected for the native and Java-to-native systems. They were all collected on the Sun boxes, which means that no JIT was used for Java. Nevertheless, the impact is not severe and is diminishing as messages are getting larger, due to the fact that the main work is done in the native library anyway. When a PvmRouteDirect was not used, the results are in the *jPVM!d* column.

Table 3 VComms Times(ms)

Bytes sent	Sun(-jit)	HP(+jit)	PC(+jit)
1	63	12	10
100	60	13	5
1000	60	13	10
10000	121	106	15
100000	236	237	115
1000000	1230	1089	1142

Table 3 presents the results collected for the VComms library on three different architectures. The performance is much slower for small messages, which means that latency values are high, as is obvious from Table 4. While we have expected inferior values on Sun due to the lack of JIT compilation, it is surprising that the results are only marginally better on HP, even though the JIT compiler was enabled. As the messages are getting larger, the differences in roundtrip times are getting smaller. Consequently the time per byte values are close.

Table 4 Communication Cost (ms)

Computers	Library	Latency	Time/Byte
Sun	LAM	0.518	9.2e-4
Sun	PVM	0.563	9.4e-4
Sun	JPVM	0.784	9.9e-4
Sun	JavaMPI	1.023	9.6e-4
Sun	VComms	81.55	0.0016
HP	VComms	51.25	0.0013
PC	VComms	5.857	0.0012

The channel throughput or bandwidth is the rate at which a network can deliver data. As a result of the high latency values, the effective bandwidth is reduced for small messages in the VComms compared to the native based libraries, by one to two orders of magnitude (Table 5).

Table 5 Bandwidth (MB/s)

Bytes	LAM	JavaMPI	Sun	HP	PC
1	0.003	9.2e-4	1.6e-5	8.3e-5	1.0e-4
100	0.19	0.129	0.002	0.007	0.02
1000	0.49	0.439	0.017	0.077	0.1
10000	1.00	0.977	0.083	0.094	0.7
100000	1.09	1.03	0.424	0.422	0.87
1000000	1.08	1.04	0.813	0.918	0.876

5.3 Discussion

While the results as presented are not very much in favor of Java it is encouraging to see the improvement due to JIT compilation. Given a system like Java, it can be expected that sending small messages is not cheap. The question remains, however, what is the cause and how can it be improved. As a starting point, we use the three steps when sending a message, as mentioned before. Assuming that the address information is stored, this is just a local call that returns an `InetAddress` object based on a group name and a host ID number. Even with no JIT, this operation takes less than a millisecond so is below the Java's timer resolution, and it can be excluded as a potential problem. Then we have to create a socket object to establish communication peers. This is a standard Java library class and its efficiency depends on the jdk. Table 6 summarizes the spread of values on our network. It is important to notice the spread, although large, is in reality much smaller since the values were concentrated at the minimum, and the maximum values were rare. The value of 0 indicates that the actual time was below timer resolution. By comparing these values with those in tables 3 and 4 it is clear that they do not dominate the roundtrip time. The reason for the best ratio being found on Sun is that the socket creation process consists mainly of native methods, and therefore is least intrusive in the interpretive environment.

Table 6 Socket Creation Time (ms)

Sun		HP		PC	
Min	Max	Min	Max	Min	Max
2	7	1	8	0	30

Once a socket is created, the data can be serialized, i.e. written into it. The serialization time presented in Table 7 represents the writing of two objects, as defined by the VComms.Send method in Section 4. Since this is mainly implemented as native code, the interpretation of bytecodes on Sun does not add much to the time. The HP results are surprising, as one would expect the values to be closer to the PC values for small messages. Similar results are obtained even when a process pings locally.

Table 7 Serialization Time (ms)

Bytes Sent	Sun		HP		PC	
	Min	Max	Min	Max	Min	Max
1	2	3	2	2	0	0
100	2	3	2	3	0	0
1000	2	3	2	3	0	0
10000	62	70	2	9	10	20
100000	145	199	209	218	90	101
1000000	1131	1213	1017	1062	922	981

Finally, we look at the receiving side of the communication channel where the serialized object is restored to its original state, but on a new process. The VComms library is implemented as a multithreaded entity, following the producer-consumer model. The sending thread makes a direct call when sending data, while the receiver is blocked at a monitor. The receiving thread, after receiving a message, adds the data to the monitor, which notifies all the receiving threads that a new data have arrived.

Table 8 Deserialization Time (ms)

Bytes Sent	Sun		HP		PC	
	Min	Max	Min	Max	Min	Max
1	54	55	12	25	0	10
100	53	55	11	16	10	10
1000	53	54	12	13	10	10
10000	115	161	77	98	10	20
100000	161	280	263	310	110	141
1000000	1155	1264	1035	1165	1312	1422

Table 8 summarizes the results, that show that for small messages most of the time is spent while deserializing the input. There are two objects to deserialize: the envelope and the content, as described in Section 4. The presented values do not include any time when the receiver was blocked waiting for input. Therefore, no thread synchronization affected the obtained values. Rather, this is just the cost of making two `VObjectInputStream.readObject` calls.

6 Conclusion

In this paper we have presented our own and related work on high performance computing and message passing in Java, and compared it to the standard native libraries like MPI and PVM. The results were, as expected, in favor of the native programming, with the Java code and native communications being the clear second. The rationale behind combining Java and native code has been in utilizing the standard communication and mathematical libraries and harvesting their better performance, while gaining on the features and flexibility provided by Java. The question remains what is the advantage of using Java in a manner that may add more complexity than C or C++ due to its features, such as the lack of pointer arithmetic and incompatible data types. Due to the different presentation, converting data from Java to native presentation and vice-versa is quite expensive and confusing due to the Java strict type checking and absence of pointer arithmetic. This explains the reasons behind our approach to aim for a pure Java parallel-processing and message-passing environment.

References

1. Dongarra, J. J., Meuer, H-W., and Strohmaier, E. The 1994 TOP500 Report. <http://www.top500.org/>
2. Geist, G. A., Beguelin, A., Dongarra, J. J., Jiang, W., Manchek, R., and Sunderam, V. S.: PVM 3 User's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory (1993)
3. Gropp, W., Lusk, E., Skjellum, A.: *Using MPI, Portable Parallel Programming with the Message-Passing Interface*. The MIT Press (1994)
4. Hsieh, C-H. A., Conte, M. T., Johnson, T. L., Cyllenhall, J. C., and Hwu, W-M. W. A Study of the Cache and Branch Performance Issues with Running Java and Current Hardware Platforms. *Proceedings of IEEE CompCon'97*, San Jose, CA. (1997) 211-216
5. JavaMPI: a Java Binding for MPI. <http://perun.hscs.wmin.ac.uk/>
6. Mangione, C. Just In Time for Java vs. C++. NC World, Vol. 7, No. 2, (January 1998).
7. MPI Forum. MPI: A Message-Passing Interface Standard, Version 1.0. (March 5, 1994). <http://www.mcs.anl.gov/mpi>
8. Ohio LAM 6.1. MPI Primer / Developing with LAM. (1996). <http://www.mpi.nd.edu/lam>
9. Stankovic, N., and Zhang, K. Visper: Parallel Computing and Java. In H.R. Arabnia (ed.) *Proceedings of the International conference on Parallel Processing Techniques and Applications*, Vol 1, CSREA Press, Las Vegas, NV (July 1998). 349-354.
10. Sun Microsystems, Inc.: Java Native Interface 1.1 Specification. (May 16, 1997) <http://java.sun.com>
11. Sun Microsystems, Inc.: Java Object Serialization Specification. Revision 1.4 (July 3, 1997) <http://java.sun.com>
12. Sun Microsystems, Inc.: Java Remote Method Invocation Specification. Revision 1.42 (October 1997) <http://java.sun.com>
13. Sun Microsystems, Inc.: Java Security Architecture (JDK 1.2). Revision 0.7 (October 1, 1997) <http://java.sun.com>
14. Thurman, D. jPVM. <http://www.isye.gatech.edu/chmsr/jPVM>

JPT: A Java Parallelization Tool

Kristof Beyls, Erik D'Hollander, and Yijun Yu

University of Ghent,
Department of Electrical Engineering
Parallel Information Systems
Sint-Pietersnieuwstraat 41
B-9000 Gent, Belgium
Kristof.Beyls@rug.ac.be
Erik.DHollander@elis.rug.ac.be
Yijun.Yu@elis.rug.ac.be

Abstract. PVM is a succesfull programming environment for distributed computing in the languages C and Fortran. Recently several implementations of PVM for Java have been added, making PVM programming accessible to the Java community.

With PVM for Java however, the user still needs to partition the problem, calculate the data partitioning and program the message passing and synchronization. In this paper, JPT is introduced, a parallelization tool which generates PVM code from a serial Java program. JPT automatically detects parallel loops and generates master and slave PVM programs.

1 Introduction

The importance of Java as a coherent, platform independent, object-oriented and network-minded language is widely recognized. With these features, it is not surprising that Java has also entered the high performance computing area with projects such as HPJava[2], JPVM[4], Java/DSM[10], Spar[8], JAVAR[1].

Most of these projects study various ways to achieve a faster execution of Java programs by efficiently expressing the parallelism in the language. Only a few authors[1] investigate the way to *automatically* detect parallel executable regions in a Java program, and to generate parallel code from this analysis. This can be attributed to the complexity of parallelization, and to the fact that parallelization tools were mainly developed for other languages[3].

In this paper we focus on the automatic parallelization and efficient code generation of Java programs. Rather than reimplementing a parallelizing compiler, the kernel parallelization algorithms and the internal syntax tree of an existing compiler, FPT[3], are reused to automatically detect parallelism in Java loops. As a result, the dependence analysis needed to reveal the parallelism in a Java program is executed by the FPT-analyzer. Once the parallel loops in the program are detected, they are transformed into an explicit parallel form by JPT. Currently, JPT generates code for 2 Java parallel platforms: parallel Java threads and jPVM.

A description of the existing parallel virtual machines for Java is given in sect. 2. The automatic parallelization of loops in the FPT parallelizer is discussed in section 3. An operational overview of JPT is given in sect. 4. The code generation is explained in sect. 5. Finally, experiments and their speedup are presented sect. 6, after which a conclusion is formulated in sect. 7.

2 Parallel Virtual Machines in Java

In the literature, there are two approaches to develop a Java based Parallel Virtual Machine: either rewrite PVM in Java[4], or write a Java-interface to the existing PVM API[7] .

1. jPVM[7] is layered upon the standard distribution of PVM and makes use of Java's capability to call functions written in other languages using the Java Native Interface[5]. jPVM programs use wrappers contained in the class jPVM to call the *native* PVM functions, which are written in C.
2. JPVM[4] on the other hand is entirely implemented in Java and uses none of the original PVM code. JPVM provides an interface similar to the C interface provided by PVM, but with a syntax and semantics adapted to Java threads and the Java programming style. Unlike jPVM, JPVM is not inter-operable with standard PVM. JPVM provides a Java implementation of the PVM daemon and a communications library. In addition, both tasks and threads are supported as basic units of parallelism.

In [9] different benchmarks were executed to test the communication performance of JPVM, jPVM and PVM. This showed jPVM to be faster than JPVM and C/PVM to be faster than jPVM. Our JPT is able to generate parallel code for the jPVM platform.

3 Loop Parallelization

3.1 Data Dependence Analysis

Loops are traditionally areas of implicit parallelism. The parallel execution of loops is subject to a non-trivial analysis of the loop-carried dependencies. Dependency analysis has matured over time and the most important dependence analysis algorithms have been put into the Fortran parallelizer, FPT[11], the backbone of JPT. FPT uses techniques derived from Banerjee, Wolfe and the GCD tests[6], loop boundary calculation and unimodular transformations[11].

By design, the inner data structures and the abstract syntax tree (AST) of FPT are language independent. As a consequence, the same dependence analysis can be applied to any language that can be expressed in the FPT syntax tree. Furthermore, the FPT API offers tools to detect, annotate and retrieve the parallelism.

3.2 Loop Scheduling

PVM code[11] for the outermost of a nest of parallel loops is obtained by generating slave programs, which each execute a group of the n iterations as one task. If there are p slaves, then n/p iterations are assigned to each slave. Besides initializing the PVM-system and contacting the number of cooperating processors, for each parallel loop the following code is generated:

- In the *prologue*, the input data for all the parallel loop is gathered and put into a single message, which is broadcast to all slave computers. Next the number of iterations to be executed by each slave is calculated and included in the message.
- In the *execution* phase, each slave program unpacks the message and executes his band of the loop. During this phase there is no communication, because the inner loop iterations are independent.
- In the *epilogue*, each slave sends back the results. The master will restore the received data in the proper locations.

3.3 Data Partitioning

After the dependence analysis, FPT determines the data to be exchanged between processors by looking for the data references to the left and the right side of assignment statements in the loop body. However, using this technique without optimization could resolve in unnecessary communication overhead. FPT uses 2 techniques to reduce the overhead:

1. If consecutive elements of an array have to be sent, then FPT will create a single *pk*-call to pack the data, instead of creating a loop in which the elements are packed one at a time.
2. The references to data in the loop might overlap. When array subscripts are of the form $ai + c$, i being the loop index and a and c are constants, FPT identifies overlapping areas and sends them only once.

When a parallel loop is part of a surrounding sequential loop, then the communication can be further optimized by *array privatization*. This technique is not yet implemented, but should work as follows. A parallel loop nest which is nested inside a sequential loop nest can be formally expressed as in fig. 1. Following sets must be calculated to create the messages between master and slaves:

- $W(i, j)$ is the set of all writes in $H(i, j)$.
- $R(i, j)$ is the set of all reads in $H(i, j)$.
- $WR(i, j)$ is the set of all reads that read a value created by a write in the same iteration.
- $E(i, j) = R(i, j) - WR(i, j)$ contains all variables and array elements that can influence the results of the parallel j loop.

```
DO i=...
  DOALL j=...
    H(i,j)
  ENDDO
ENDDO
```

Fig. 1. A parallel loop inside a sequential loop

The data to be sent between iteration i and $i + 1$ of the sequential loop from task p_1 to task p_2 would then be $S(i \rightarrow i + 1, p_1, p_2) = W_{p_1}(i) \cap E_{p_2}(i + 1)$, with $W_{p_1}(i) = \bigcup_{j \in I(p_1)} W(i, j)$ and $E_{p_2}(i) = \bigcup_{j \in I(p_2)} E(i, j)$ where $I(p)$ is the set of iterations to be executed by task p .

4 JPT Operational Overview

The conversion of a Java program into an FPT AST and the parallelism extraction occurs in four steps (see Fig.2):

1. The Java source is *parsed* using the GNU compiler **guavac** into a complete Java-based abstract syntax tree (in this paper further called a Guavac AST). Since FPT was developed for Fortran, obviously some Java language constructs cannot be represented by the abstract syntax tree of FPT. However, the computation intensive parts, most amenable to parallelization, are represented similarly in both languages, i.e. by loops and array calculations. As a consequence, only a part of the Guavac AST is transformed into an FPT AST.
2. JPT *translates* the parts in the Guavac AST that are expressible in FPT and feeds them one by one into the *parallelizer* of FPT.
3. The resulting parallelized FPT AST is traversed to see which *loops* were *parallelized* by FPT, and the corresponding loops in the Guavac AST are marked as parallelizable. The FPT parallelizer also generates the messages to be sent between master and slaves.
4. explicit *parallel code is generated* from the annotated Guavac AST. Currently JPT generates parallel code based on
 - (a) Java threads,
 - (b) jPVM.

5 Code Generation

JPT transforms the original program into an explicitly parallel PVM program by replacing each parallel loop by a master loop which calls a number of slaves and fetches the results.

A separate class is inserted to contain the slave code. This class extends the JPT. **JptPvmSlave** class (see Fig. 3(a)) which implements the job scheduling code common to all slaves. A **run** method is created in the new class. The loop specific slave code generated in the next steps will be inserted, in this **run** method. The interaction between the master and slaves is as follows:

1. The *master* spawns the slaves using the method **spawn_slaves** (see Fig. 3(b)). The **spawn_slaves** method spawns a new Java Virtual Machine for each slave, after which it sends the job information to the newly spawned slaves.

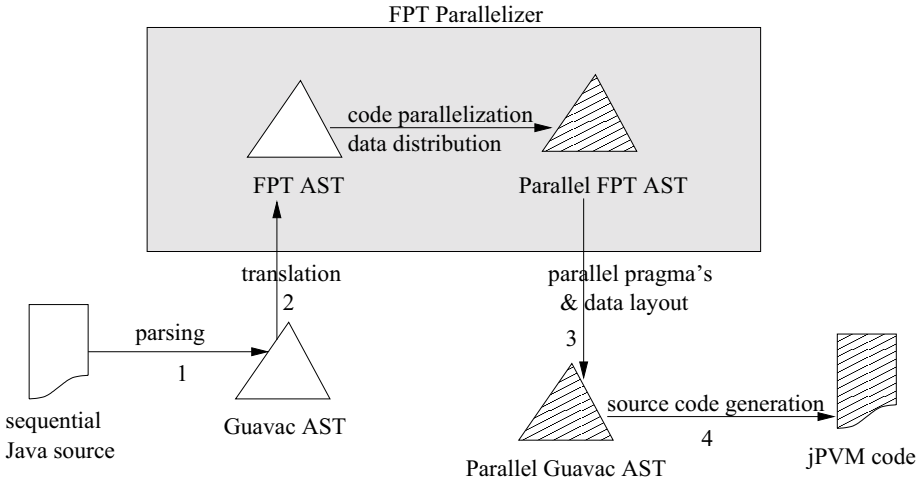


Fig. 2. JPT Parsing, Parallelization and Code Generation. The source file is parsed into an Abstract Syntax Tree by Guavac. The loop nests are forwarded to FPT. After parallelization by FPT, the parallel loops are annotated in the Guavac AST. Finally, the code can be generated for different parallel Java platforms.

2. Each JVM will execute one *slave*. The spawned JVM knows which slave to execute because the name of the slave class is passed as a command line argument. The spawned JVM will execute the `main` method of the `JptPvmSlave` class (see Fig. 3(a)). The `main` method creates an object of the slave class, and lets the slave execute in a separate thread. When the slave thread stops, the JVM ends executing. Creating an object of the slave class forces the constructor (starting at `/* 2 */` in Fig. 3(a)) to be executed. This constructor receives the job information sent in the `spawn_slave` method at `/* 1 */`. Each task is run in a separate JVM because PVM and therefore jPVM are not thread-safe (see section 2).
3. After spawning the slaves, the *master* code packs the sizes of all the arrays in the parallel loop as well as the input data for all iterations of the loop into a single message and multicasts it to all slaves.
The slave code (which is put in the `run()` method) receives this message, creates the arrays, and executes his band of the loop.
4. The *slave* sends the output data for that band to the master.

6 Results

The performance of the code generated by JPT was measured using a matrix multiplication and a Gauss-Jordan linear system solver. The tests were performed on 4 Pentium-II machines interconnected with a 100 Mb/s Ethernet running Linux as operating system and using JDK 1.1.7 as the Java Virtual Machine.

```

package JPT;
import jPVM;

public abstract class JptPvmSlave
    implements Runnable
{
    public int mytid=-1;
    public int parent_tid=-1;
    public int nslaves=-1;
    public int[] tids;
    public int slavenum=-1;
    public int band=0;

    /* 2: job information is received */
    // constructor initializes job
    public JptPvmSlave()
    {
        mytid=jPVM.mytid();
        parent_tid=jPVM.parent();

        // PVM -> Java data receive
        jPVM.recv(parent_tid,STARTUP_MESG);
        int[] nslaves_band=new int[2];
        jPVM.upkint(nslaves_band,2,1);
        nslaves=nslaves_band[0];
        band=nslaves_band[1];
        tids=new int[nslaves];
        jPVM.upkint(tids,nslaves,1);
        for(int i=0; i<tids.length; i++)
            if (tids[i]==mytid) {
                slavenum=i;
                break;
            }
    }

    // slave entry point
    public static void main(String[] args)
    throws ClassNotFoundException,
        InstantiationException,
        IllegalAccessException,
        InterruptedException
    {
        Class thisClass =
            Class.forName(args[0]);
        JptPvmSlave thisClassInstance =
            (JptPvmSlave)
                thisClass.newInstance();
        Thread t = new
            Thread(thisClassInstance);
        t.start();
        t.join();
    }
}

```

(a) The base class JptPvmSlave in Java package JPT

```

public static int spawn_slaves
(int[] tids, String slavename, int band)
{
    final int nrequested = tids.length;
    int mytid = jPVM.mytid();

    String[] args=new String[2];
    args[0]=slavename;
    args[1]=slavename;
    // spawn slaves
    if (tids.length >
        jPVM.spawn("java", args,
                    jPVM.PvmDataDefault,
                    "",tids))
    {
        System.err.println(
            "in JPT.JptPvmSlave.spawn_slaves"
            +"(int[] tids): couldn't spawn "
            +"the number of tasks requested"
        );
        jPVM.exit();
        System.exit(-2);
    }

    /* 1: Job information is sent */
    // Java -> PVM data send
    jPVM.initsend(jPVM.PvmDataDefault);
    int[] ntasks_band=new int[2];
    ntasks_band[0]=tids.length;
    ntasks_band[1]=band;
    jPVM.pkint(ntasks_band,2,1);
    jPVM.pkint(tids,tids.length,1);
    jPVM.mcast(tids,STARTUP_MESG);
    return mytid;
}

```

(b) The spawn_slaves method

Fig. 3. The JPT package

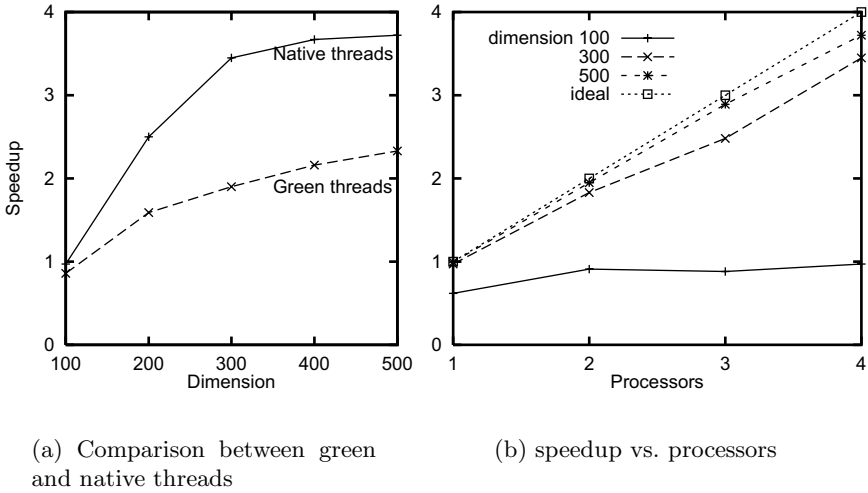


Fig. 4. Speedups of the matrix multiplication algorithm

First the parallelized matrix multiplication algorithm was executed using "green" threads, i.e. threads scheduled within the Java Virtual machine. In this implementation the operating system executes the JVM as a whole, and has no grip on load balancing the individual threads. As a result, some long idle periods during the inter process communication were observed, affecting the overall speedup. In a second experiment, native threads were used, i.e. the Java threads are visible as individual threads to the operating system. This resulted in a better scheduling and good communication and speedup figures. Figure 4(a) depicts the speedup difference between native and green threads. Using native threads, the speedup has been measured for 1 to 4 processors, yielding a fairly linear speedup for a dimension > 300 . (see fig. 4(b)).

We further tested the results on the Gauss-Jordan Elimination algorithm. We found that the data distribution as calculated by FPT creates excessive data communication. When this algorithm is transformed using array privatization, and the data communication is done as resulted from the proposed technique in sect. 3.3, good speedups were found, as seen in fig. 5.

7 Conclusion

The Java parallelizer JPT facilitates the use of Java in a PVM environment. The results show that loops can be parallelized using standard techniques embedded in the open compiler FPT, including data distribution and message coding. Further work is aimed at reducing the communication overhead by a sophisticated array privatization analysis. The speedups obtained indicate that Java is a viable language for parallel computing in a platform independent PVM-network.

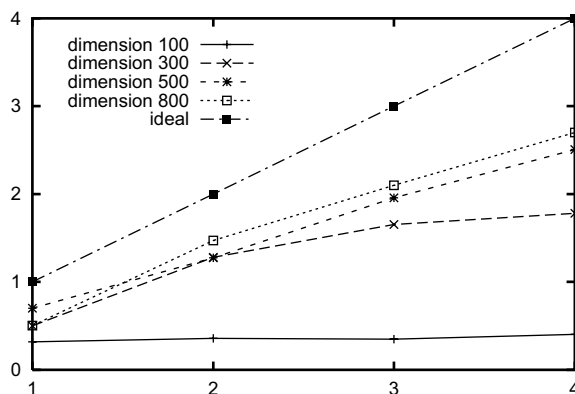


Fig. 5. Speedup of the optimized Gauss-Jordan linear system solver.

References

- [1] A. J. C. Bik, J. E. Villacis, and D. B. Gannon. javar: a prototype Java restructuring compiler. *Concurrency, Pract. Exp. (UK), Concurrency: Practice and Experience*, 9(11):1181–1191, Nov. 1997.
- [2] B. Carpenter, G. Zhang, G. Fox, X. Li, and Y. Wen. HPJava: Data parallel extensions to java. *Concurrency: Practice and Experience*, 10(11-13):873–877, 1998.
- [3] E. D'Hollander, F. Zhang, and Q. Wang. The fortran parallel transformer and its programming environment. *Journal of Information Sciences*, 106(7):293–317, July 1998.
- [4] A. J. Ferrari. JPVM: Network parallel computing in java. In *Proceedings of the ACM Workshop on Java for High-Performance Network Computing*, Mar. 1998.
- [5] JavaSoft. Java native interface specification, Nov. 1996. Release 1.1.
- [6] K. Psarris. The Banerjee-Wolfe and GCD tests on exact data dependence information. *Journal of Parallel and Distributed Computing*, 32(2):119–138, Feb. 1996.
- [7] D. Thurman. jPVM. <http://www.isye.gatech.edu/chmsr/jPVM/>.
- [8] K. van Reeuwijk, A. J. van Gemund, and H. J. Sips. Spar: A programming language for semi-automatic compilation of parallel programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, Nov. 1997.
- [9] N. Yalamanchilli and W. Cohen. Communication performance of java based parallel virtual machines. *Concurrency: Practice and Experience*.
- [10] W. M. Yu and A. L. Cox. Java/DSM: a platform for heterogeneous computing. In *Proc. of Java for Computational Science and Engineering-Simulation and Modeling Conf.*, pages 1213–1224, June 1997.
- [11] F. Zhang. *The FPT Parallel Programming Environment*. PhD thesis, University of Ghent, 1996.

Facilitating Parallel Programming in PVM Using Condensed Graphs *

John P. Morrison and Ronan W. Connolly

University College Cork, Ireland

`j.morrison@cs.ucc.ie`, `r.connolly@cs.ucc.ie`

`http://www.cuc.ucc.ie`

Abstract. This paper describes the implementation of the Condensed Graph (\mathcal{CG}) Computing Model using the PVM system. This model enables the programmer to write solutions to problems to run on a PVM System without the programmer having to create a parallel solution or embed PVM library routines into their program. This paper describes the basic concepts behind the \mathcal{CG} Model. It then describes how the \mathcal{CG} Model is implemented using the PVM System.

1 Introduction

In general, programming parallel systems is complicated by the fact that the programmer is responsible for explicitly managing the synchronization and sequencing of tasks. Earlier work in data-driven computing [3,2,5] addressed this problem but with limited success: granularity was too fine and there was no natural mechanism for handling data structures. Subsequent work including [4,7,11,16] addresses these problems with more success. Work has also been done in facilitating parallel programming by providing the user with a graphical programming environment [6,15].

The Condensed Graphs model employs a variable grain of execution, can sequence instructions imperatively, eagerly and lazily, and although not discussed further here, naturally incorporates data structures.

In conjunction with the PVM system [8], the implementation of the Condensed Graphs model acts as a high-level parallel scheduler which relieves the programmer from the burdens described above.

2 Condensed Graphs

While being conceptually as simple as classical dataflow schemes [9,1], the Condensed Graphs (\mathcal{CG}) model is far more general and powerful [14,12]. It can be described concisely, although not completely, by comparison. Classical dataflow is based on data dependency graphs in which nodes represent operators and

* Research Grant aided by the National Software Directorate, Ireland.

edges are data paths which convey simple data values between them. Data arrive at *operand ports* of nodes along input edges and so trigger the execution of the associated operator (in dataflow parlance, they cause the node to *fire*). During execution, these data are consumed and a resultant datum is produced on the node's outgoing edges. This result acts as input to successor nodes. In cyclic dataflow graphs, nodes may fire more than once and so operands belonging to different executions are distinguished from each other using labels. When data with the same label are present on every operand port, a *complete operand set* is formed. Operand sets are used at the basis of the firing rules in data-driven systems. These rules may be *strict* or *non-strict*. A strict firing rule requires a complete operand set to exist before a node can fire; a non-strict firing rule triggers execution as soon as a specific proper subset of the operand set is formed. The latter rule gives rise to more parallelism but also can result in overhead due to remaining packet garbage (RPG).

Like classical dataflow, the \mathcal{CG} model is graph-based and uses the flow of entities on arcs to trigger execution. In contrast, \mathcal{CG} s are directed acyclic graphs in which every node contains not only operand ports, but also an operator and a destination port. Arcs incident on these respective ports carry other \mathcal{CG} s representing operands, operators and destinations. Condensed Graphs are so called because their nodes may be condensations, or abstractions, of other \mathcal{CG} s. (Condensation is a concept used by graph theoreticians for exposing meta-level information from a graph by partitioning its vertex set, defining each subset of the partition to be a node in the condensation, and by connecting those nodes according to a well-defined rule [10].) Condensed Graphs can thus be represented by a single node (called a *condensed node*) in a graph at a higher level of abstraction. The number of possible abstraction levels derivable from a specific graph depends on the number of nodes in that graph and the partitions chosen for each condensation. Each graph in this sequence of condensations represents the same information but in a different level of abstraction. It is possible to navigate between these abstraction levels, moving from the specific to the abstract through condensation, and from the abstract to the specific through a complementary process called *evaporation*.

The basis of the \mathcal{CG} firing rule is the presence of a \mathcal{CG} in every port of a node. That is, a \mathcal{CG} representing an operand is associated with every operand port, an operator \mathcal{CG} with the operator port and a destination \mathcal{CG} with the destination port. This way, the three essential ingredients of an instruction, also called an *instruction triple*, are brought together (these ingredients are also present in the dataflow model; only there, the operator and destination are statically part of the graph).

A condensed node, a node representing a datum, and a multi-node \mathcal{CG} can all be operands. A node represents a datum with the value on the *operator* port of the node. Data are then considered as zero-arity operators. Datum nodes represent graphs which cannot be evaluated further and so are said to be in *normal form*. Condensed node operands represent unevaluated expressions. They cannot be fired since they lack a destination. Similarly, multi-node \mathcal{CG} operands

represent partially evaluated expressions. The processing of condensed node and multi-node operands is discussed below.

Any \mathcal{CG} may represent an operator. It may be a condensed node, a node whose operator port is associated with a machine primitive (or a sequence of machine primitives) or it may be a multi-node \mathcal{CG} .

The present representation of a destination in the \mathcal{CG} model is as a node whose own destination port is associated with one or more port identifications. The expressiveness of the \mathcal{CG} model can be increased by allowing any \mathcal{CG} to be a destination but this is not considered further here. Fig. 1 illustrates the congregation of instruction elements at a node and the resultant rewriting that takes place.

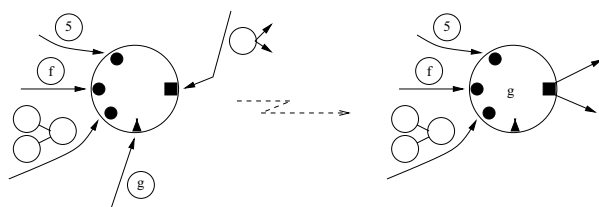


Fig. 1. \mathcal{CG} s congregating at a node to form an instruction.

When a \mathcal{CG} is associated with every port of a node it can be fired. Even though the \mathcal{CG} firing rule takes account of the presence of operands, operators and destinations, it is conceptually as simple as the dataflow rule. Requiring that the node contain a \mathcal{CG} in every port before firing prevents the production of RPG. As outlined below, this does not preclude the use of non-strict operators or limit parallelism.

A *grafting* process is employed to ensure that operands are in the appropriate form for the operator: non-strict operators will readily accept condensed or multi-node \mathcal{CG} s as input to their non-strict operands. Strict operators require all operands to be data. Operator strictness can be used to determine the strictness of operand ports: a strict port must contain a datum \mathcal{CG} before execution can proceed, a non-strict port may contain any \mathcal{CG} . If, by computation, a condensed or multi-node \mathcal{CG} attempts to flow to a strict operand port, the *grafting* process intervenes to construct a destination \mathcal{CG} representing that strict port and sends it to the operand.

The grafting process thus facilitates the evaluation of the operand by supplying it with a destination and, in a well constructed graph, the subsequent evaluation of that operand will result in the production of a \mathcal{CG} in the appropriate form for the operator. The grafting process, in conjunction with port strictness, ensures that operands are only evaluated when needed. An inverse process called *stemming* removed destinations from a node to prevent it from firing.

Strict operands are consumed in an instruction execution but non-strict operands may be either consumed or propagated. The \mathcal{CG} operators can be divided into two categories: those that are ‘value-transforming’ and those that only move \mathcal{CG} s from one node to another in a well-defined manner. Value-transforming operators are intimately connected with the underlying machine and can range from simple arithmetic operations to the invocation of sequential subroutines and may even include specialized operations like matrix multiplication. In contrast, \mathcal{CG} moving instructions are few in number and are architecture independent. Two interesting examples are the condensed node operator and the **filter** node. Filter nodes have three operand ports: a Boolean, a **then** and an **else**. Of these, only the Boolean is strict. Depending on the computed value of the Boolean, the node fires to send either the **then** \mathcal{CG} or the **else** \mathcal{CG} to its destination. In the process, the other operand is consumed and disappears from the computation. This action can greatly reduce the amount of work that needs to be performed in a computation if the consumed operands represent an unevaluated or partially evaluated expression. All condensed node operators are non-strict in all operands and fire to propagate all their operands to appropriate destinations in their associated graph. This action may result in condensed node operands (representing unevaluated expressions) being copied to many different parts of the computation. If one of these copies is evaluated by grafting, the graph corresponding to the condensed operand will be invoked to produce a result. This result is held local to the graph and returned in response to the grafting of the other copies. This mechanism is reminiscent of parallel graph reduction [17] but is not restricted to a purely lazy framework.

\mathcal{CG} s which evaluate their operands and operator in parallel can easily be constructed by introducing **spec** (speculation) nodes to act as destinations for each operand. The **spec** node has a single operand port which is strict. The multi-node \mathcal{CG} operand containing the **spec** node is treated by non-strict operand ports in the same way as every other \mathcal{CG} , however, if it is associated with a strict port, the **spec** node’s operand is simply transferred to that port. If that operand had already been fully evaluated, it could be used directly in the strict port, otherwise, it is grafted onto the strict port as described above. This is illustrated in Fig. 2.

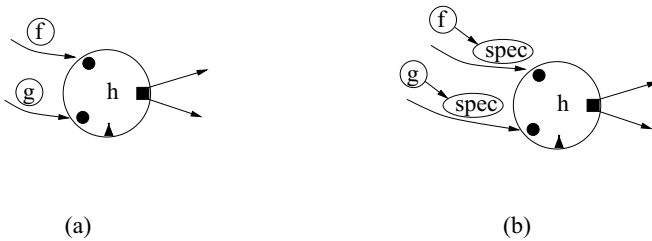


Fig. 2. *Increasing parallelism by speculatively evaluating operands.*

By statically constructing a \mathcal{CG} to contain operators and destinations, the flow of operand \mathcal{CG} s sequences the computation in a dataflow manner. Similarly, constructing a \mathcal{CG} to statically contain operands and operators, the flow of destination \mathcal{CG} s will drive the computation in a demand-driven manner. Finally, by constructing \mathcal{CG} s to statically contain operands and destinations, the flow of operators will result in a control-driven evaluation. This latter evaluation order, in conjunction with side-effects, is used to implement imperative semantics. The power of the \mathcal{CG} model results from being able to exploit all of these evaluation strategies in the same computation, and dynamically move between them, using a single, uniform, formalism.

3 Managing Instruction Triples

An initial \mathcal{CG} evaluator, implemented to construct and subsequently evaluate instruction triples is illustrated in Fig. 3. This Triple Manger (TM) uses a memory to hold evaluating graphs, tables to define instructions and queues to hold instructions and results.

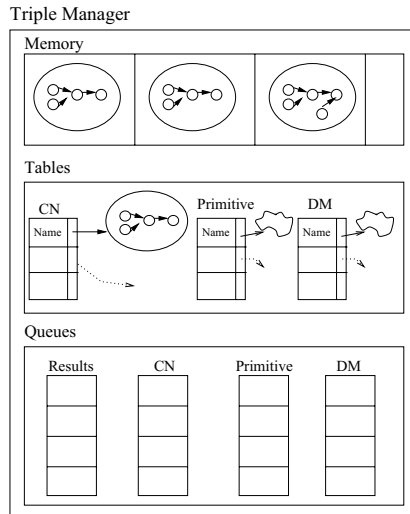


Fig. 3. Structure of the Condensed Graphs Triple Manager.

For convenience, each instruction kind is held in its own separate queue. The top-level functionality of the TM involves the following loop: determine all executable instructions and place them in their respective queues; execute instructions in each queue; process all results; deallocate results if possible.

Executing Data Moving (DM) Instructions: Executing a DM instructions results in moving condensed nodes around in memory. Apart from **filter** nodes, one

prevalent *DM* instruction is the **X** (for exit) node. By construction this is the last node in each graph. When executed it sends a resultant condensed node back to the destination of its parent node and subsequently marks its own graph for deallocation. This marking scheme in conjunction with an *outstanding result counter* is employed, in favour of direct deallocation, so that any results in transit, destined for this graph can be handled – direct deallocation would have left these results orphaned.

Executing Primitive Instructions: These value transforming instructions are invoked indirectly through the **Primitive Table** and results are placed on the **Results Queue**.

Executing Condensed Nodes: The execution of a condensed node results in the allocation of memory to hold its corresponding \mathcal{CG} , the placing of the condensed node’s operands appropriately into that graph, and the sending of the condensed node’s destination to the **X** node of the graph. Executing condensed nodes gives rise to parallelism since more instructions are made available for execution. Conversely, the execution of **X** nodes reduces the amount of parallelism by eliminating graphs from memory. The important point here is that by mixing evaluation orders it is possible for an **X** node to fire even if there are still executable instructions in its containing graph.

By their nature *DM* instructions must be executed by the *TM* on which they originate, however, primitive and condensed node instructions can be dispatched for execution elsewhere. To facilitate this a **Distributed Triple Manager** (*DTM*), as illustrated in Fig. 4, is employed.

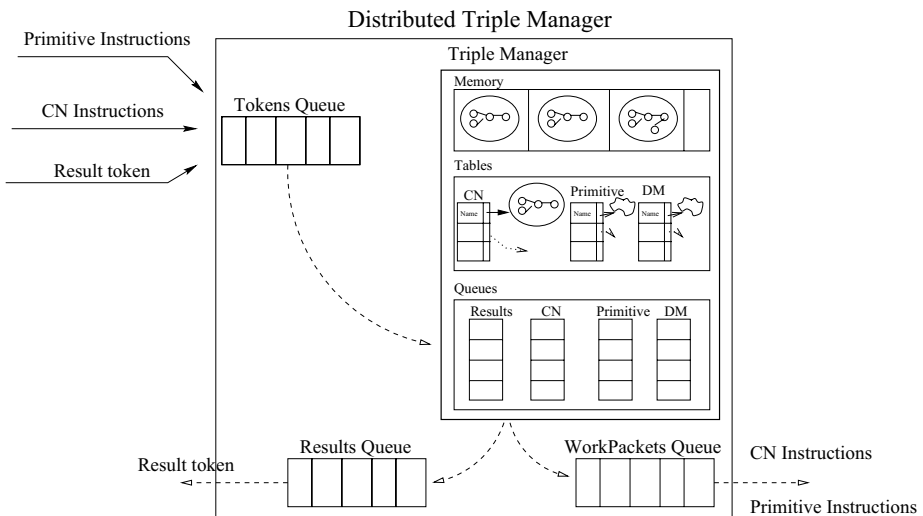


Fig. 4. *Structure of the Distributed Triple Manager.*

In addition to the *TM* the *DTM* contains 3 queues to support the communication of results and instructions.

When the decision is taken to evaluate a primitive or condensed node instruction remotely, a **WorkPacket** is constructed and placed in the **WorkPackets Queue**.

A **WorkPacket** consists of **<Token, result_destination>** where **Token** is **<SenderId, WorkPacketID, TokenKind>**. The **SenderId** is the process identification of the sender, the **WorkPacketID** is the index value of the **WorkPacket** in the **WorkPacket Queue** and the **TokenKind** is either a primitive or a condensed node instruction. The **result_destination** in the **WorkPacket** identifies the destination of the result associated with the instruction of the **Token**. When this token is remotely executed, a **Result Token** is subsequently returned, its **WorkPacketID** is used to extract a destination from the **Workpacket Queue** and the corresponding **Workpacket** is deleted.

Tokens incident on the *DTM* can be either results from a remotely executed instruction or a request to execute either a primitive or *CN* instruction.

A result token is unpacked as discussed above. The appropriate *TM* memory location for this result is determined from the appropriate **WorkPacket** in the **Results Queue**.

When processing a primitive instruction token, a result token is partially constructed and placed in the **Results Queue**. The primitive table of the *TM* is then invoked to yield a resultant value. This value is then placed in the partially constructed result token – ready for subsequent dispatch.

Execution of a *CN* instruction is somewhat more complex but similar in format. A partially constructed result is placed in the **Results Queue** as before. The *CN* is then placed into the **CN Queue** of the *TM*. This entry is annotated to reflect the fact that the *CN* instruction originated from another *TM*. All *CN* instructions are executed in the same way by invoking the **CN Table** to allocate a new graph instance in the *TM* Memory. However the **X** node of the remotely invoked *CN* is mutated so that the result, as done with primitive instructions, is placed into the **Result Queue** as opposed to being placed back into *TM* memory. The results from the **Results Queue** can be subsequently dispatched.

In addition to supporting the basic operation of the *DTM*, the **Token**, **WorkPackets** and **Results** queues can be used for supporting load-balancing and fault tolerance.

4 Conclusions and Ongoing Work

The implementation discussed here is currently being tested on a 16 node Beowulf cluster. Optimum node granularity is being determined and various fault tolerant and load-balancing strategies are being investigated. The model has already been implemented on the WWW and encouraging results for programs containing sufficient granularity have been obtained [13].

References

1. Arvind and Kim P. Gostelow. A Computer Capable of Exchanging Processors for Time. Information Processing 77 Proceedings of IFIP Congress 77 Pages 849-853, Toronto, Canada, August 1977.
2. Arvind, David E. Culler and Kattamuri Ekanadham. The Price of Asynchronous Parallelism: An Analysis of Dataflow Architectures. Conpar '88, Pages 168-182, Manchester, England 1998.
3. D. Ambramson and G. Egan. The RIMT Data Flow Computer: A Hybrid Architecture. The Computer Journal, 33(3):230-240, June 1990.
4. Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu and Willy Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations IEEE Computer Vol 29, No. 2, pages 18-28, Feb 1996.
5. J. Backus. Can Programming be Liberated from the Von Nuemann Style? A Functional Style and its Algebra of Programs. Communications of the ACM, 21(6):613-641, August 1978.
6. A. Beguelin, J. Dongarra, A. Geist, R. Manchek, K. Moore and V. Sunderam. PVM and HeNCE: Tools for Heterogeneous Network Computing. Software for parallel computation: Proceedings of the NATO Advanced Workshop on Software for Parallel Computation, held at Cetraro, Cosenza, Italy, June 22-26, 1992.
7. Robert D. Blumofe and Christopher F. Joerg and Bradley C. Kuszmaul and Charles E. Leiserson and Keith H. Randall and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. Proc. 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP'95. Santa Barbara, California, July 1995.
8. Al Geist, Adam Beguelin, Jack Dongarra, Weich Jiang, Robert Manchek, and Vaidy Sunderam. PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, 1994.
9. J.R. Gurd, C.C. Kirkham, and Ian Watson. The Manchester Prototype Data Flow Computer. Communications of The ACM, 28(1):34-52, January 1985.
10. Frank Harary, Robert Norman and Dorwin Cartwright. Structural Models: An Introduction to the Theory of Directed Graphs John Wiley and Sons, 1969.
11. Keneth R. Traub, Gregory M. Papadopoulos, Michael J. Beckerle, James E. Hicks, Jonathan Young. Overview of the Monsoon Project. Computation Structures Group Memo 339, January 1991.
12. John Morrison, Martin Rem. Managing and Exploiting Speculative Computations in a Flow Driven, Graph Reduction Machine. PDPTA '99. Las Vegas, USA.
13. John Morrison, David Power, James Kennedy. A Condensed Graphs Engine to Drive Metacomputing. PDPTA '99. Las Vegas, USA.
14. John P. Morrison: Condensed Graphs: Unifying Availability-Driven, Coercion-Driven and Control-Driven Computing. ISBN: 90-386-0478-5. Technische Universiteit Eindhoven, October 1996. (PhD Thesis)
15. Peter Newton and Jack Dongarra. Overview of VPE: A Visual Environment for Message-Passing, Parallel Programming. Technical Report University of Tennessee CS-94-261 November 1994.
16. Rishiyur Nikhil, Gregory M. Papadopoulos, Arvind. *T: A Multithreaded Massively Parallel Architecture. Computation Structures Group Memo 325-2, March 5, 1992.
17. Rinus Plasmeijer and Marko van Eekelen. Functional Programming and Parallel Graph Reduction ISBN: 0-201-41663-8 Addison-Wesley Publishers Ltd.

Nested Bulk Synchronous Parallel Computing

F. de Sande, C. León , C. Rodríguez, J. Roda, J.A. González

Departamento de Estadística, I. O. y Computación
Universidad de La Laguna
Facultad de Matemáticas. c/ Astrofísico Francisco Sánchez, s/n
38271 La Laguna S/C de Tenerife
SPAIN
sande@csi.ull.es, cleon@ull.es, casiano@ull.es

Abstract. The BSP model can be extended with the inclusion of processor sets. In this model, processor sets can be divided and processor sets synchronize through Collective operations. The study of optimal policies for the implementation of the Division Functions lead us to the concept of dynamic polytope. The advantages of the model are exemplified through the divide and conquer paradigm. Computational results for two instances of this paradigm in three parallel machines are presented.

1. Introduction

The Bulk Synchronous Parallel (BSP) model is a generalization of the widely researched PRAM model initially proposed by Valiant [1]. Although the initial formulation of BSP considered the possibility of machine decomposition as also did the BSP library standard proposal, they decided in the absence of any clear solution to exclude it from the standard. In this paper we present some proposals for a Nested Bulk Synchronous Parallel Model (NBSP), that is, to extend BSP with the inclusion of processor sets. Processor sets can be divided using what we call Division Functions and processors can perform Collective operations implying all the processors in the set. Like in BSP, computation takes place in steps bounded by the execution of collective operations.

The remainder of the paper is organized as follows: section 2 introduces the NBSP model, stating its characteristics. The properties that the family of Division and Collective Functions have to hold are defined in that section. The laws that govern the time-cost of program execution under the model are also introduced. The third section proposes strategies for the implementation of Division Functions. Experimental results for a Cray T3E, a Cray T3D and an IBM SP2 for the Fast Fourier Transform and the Quicksort algorithm are presented and discussed in section 4. The results show that there are opportunities for improvement in current implementations of Division Functions, like those provided by MPI. Section 5 presents the conclusions.

2. The Nested BSP Computing Model

Let us consider a parallel computer made of a set $P = \{p_0, p_1, \dots, p_{N-1}\}$ of memory-processor units connected through some kind of interconnection network. At any time, the processors are organized in a *hierarchy of processor sets*. At the beginning of the computation all the processors P of the machine belong to the *root* set of processors. All processors in a set execute the same *task*. Each processor in a *processor set* has two private variables associated, which will be referred as *NAME* and *NUMPROCESSORS*. While each processor in the *set* has a different *NAME* value, all of them share a common value for the private variable *NUMPROCESSORS*. This last value indicates the cardinality of its processor set.

The Nested Bulk Synchronous Parallel Model (NBSP) is determined by the tuple $(N, Div, Col, (g_r)_{r \in \{1, \dots, N\}}, (L_r)_{r \in \{1, \dots, N\}})$. N is the number of processors. Div and Col are, respectively, the set of division and collective functions. In the NBSP, the communication power of a parallel machine is characterized by a family of bandwidth $(g_r)_{r \in \{1, \dots, N\}}$ and latency $(L_r)_{r \in \{1, \dots, N\}}$ parameters, where the index r varies in the range of processor numbers $\{1, \dots, N\}$. Values g_r and L_r are the bandwidth and latency of any submachine of size r . The model can be simplified to take as equal all the g_r and L_r values. At any time, the processors are constrained to execute one of these three alternatives:

1. Any kind of sequential computation: assignments, loops, function calls, etc.
2. Operations $D \in Div$
3. Operations $C \in Col$

Let $Q \subseteq P$ be a set of processors executing a *task* T . Such an executing set Q will be referred to as a *leaf set*. The Division Functions have to be executed by *all* the processors in a leaf set Q . When the processors in Q execute a division function $D \in Div$:

$$D(T_0(in_0, out_0), in_0, out_0, T_1(in_1, out_1), in_1, out_1, \dots, T_r(in_{r-1}, out_{r-1}), in_{r-1}, out_{r-1})$$

the following sequence of events has to occur:

1. The *leaf set* Q is partitioned in r disjoint subsets Q_0, \dots, Q_{r-1} :

$$Q = \cup_{0 \leq i \leq r-1} Q_i ; Q_i \cap Q_j = \emptyset \text{ if } i \neq j \quad \forall 0 \leq i, j \leq r-1$$

On each processor $q \in Q_i$ the variable *NUMPROCESSORS* is updated to $|Q_i|$ and variable *NAME* will hold consecutive and different values for the processors in Q_i . The processors in subset Q_i are assigned to the parallel task $T_i(in_i, out_i)$. In_i and Out_i are the input and output data of task T_i respectively.

2. Assign the input data in_i corresponding to the parallel task T_i to the processors in Q_i . After this step, the processor set Q becomes an inner node in the processor hierarchy and the new subsets Q_i will be leaf sets.

3. Execute the parallel tasks $T_i(in_i, out_i)$. Processors in Q_i perform the task T_i whose input data are in_i and whose results are out_i . From this point on, the term *task* will refer to each of the parallel processes T_i appearing as parameters in a division function D .

4. At the end of the execution of all the tasks T_i the subsets Q_i will join again to form the set Q . The output data out_i generated by the parallel tasks T_i will be distributed among the processors.

The third component in the tuple defining the Nested BSP Computing Model is the group of collective functions Col . A collective operation $C \in Col$ performed by a leaf set of processors $Q = \{p_0, p_1, \dots, p_{k-1}\} \subseteq P$ is an operation $C(in_0, out_0, \dots, in_{k-1}, out_{k-1})$ on a set of input data that has to be executed by all the processors p_i in Q .

Computation in the NBSP occurs in steps that will be referred to as supersteps, following the BSP terminology [3]. The cost $\Phi(T(in, out), N, g_N, L_N)$ of a NBSP program T executed by a set with N processors, bandwidth g_N and latency L_N is obtained as the sum of the costs Φ_s of its supersteps. We consider two kinds of supersteps. The first kind, called *normal* superstep has two stages:

1. Local computation and asynchronous messages or remote memory accesses.
2. Execution of a collective communication function C from Col .

As in BSP, the asynchronous communications and remote memory accesses performed during the first stage are made effective only at the end of the superstep. The second communication stage can be simply a processor subset barrier.

The cost Φ_s of a *normal* superstep s is given by:

$$\Phi_s = W + T_C = \max \{W_i / i = 0, \dots, k-1\} + (g_k * h + L_k)$$

The first term (W) represents the computational cost, and T_C is the term associated with communications. $k = |Q|$ is the cardinal of the processor subset Q executing this superstep and W_i is the time invested in computation by processor i in this superstep. The number h is the h -relation of the superstep: the maximum number of messages communicated by any processor in Q , including those associated with the call to the collective function.

The second kind of superstep defined in the model is the *division* superstep. This *superstep* occurs when a division function $D \in Div$ is executed. The time or cost Φ_s is given by:

$$\begin{aligned} \Phi_s = & (g_k * h_{D,in} + L_k) + (g_k * h_{D,out} + L_k) + \max \{W_{D,i} / i = 0, \dots, k-1\} + \\ & + \max \{ \Phi(Task_0(in_0, out_0), P_0, g_{P_0}, L_{P_0}), \dots, \Phi(Task_{r-1}(in_{r-1}, out_{r-1}), P_{r-1}, g_{P_{r-1}}, L_{P_{r-1}}) \} \end{aligned}$$

where $h_{D,in}$ is the (optional, some division functions omit this phase) h -relation produced by the input data distribution and $h_{D,out}$ corresponds to the (also optional) resulting data interchange associated with D . The value $W_{D,i}$ is the computing time spent by processor i in the division and reunification. The value P_i is the number of processors assigned by D to $Task_i(in_i, out_i)$ that is, $P_i = |Q_i|$. The last term is the maximum of the times $\Phi(Task_i(in_i, out_i), P_i, g_{P_i}, L_{P_i})$ recursively computed following the NBSP model for the r subsets.

3. Strategies for the Implementation of Division Functions

Although the need of Division Functions appears in a wide class of algorithms, no doubt Divide and Conquer algorithms constitute a incentive for the introduction and

formalization of Division Functions. The Nested BSP Computing Model makes easier the implementation of *common-common* Divide and Conquer algorithms. A problem to be solved in parallel is said to be a *common-common problem* if, initially, the input data are replicated in all the processors and at the end, the solution to the problem is required to be in all the processors as well. The divide and conquer approach presented in Fig. 1 to find the solution of a problem x proceeds by dividing a common-common problem x in subproblems x_0 and x_1 (function *divide* in line 6) and recursively applying the same resolution scheme.

```

1  procedure pDC(x: Problem; r: Result);
2  begin
3    if trivial(x) then conquer(x, r)
4  else
5    begin
6      divide(x, x0, x1);
7      PARALLEL(pDC(x0, r0), x0, r0, pDC(x1, r1), x1, r1);
8      combine(r, r0, r1);
9    end;
10 end;
```

Fig. 1. General frame for a parallel divide and conquer algorithm

The call to the Division Function $\text{PARALLEL} \in \text{Div}$ on line 7 of the code produces the parallel activation of two tasks (pDC) to solve each of the two subproblems in which the original problem has been divided. The PARALLEL Division Function divides the actual set of leaf processors into two subsets. Each of the subsets solves a sub-problem x_i in parallel, and at the end of the division process, all processors in the original set achieve the solution r by combining the partial solutions r_0 and r_1 .

3.1 A Scheme for the Implementation of Division Functions in *Common-Common* Divide and Conquer Algorithms

Let $\Gamma = \{Q_0, \dots, Q_{m-1}\}$ be a partition of a set Q . Sets Q_j with $j \neq i$ will be referred to as complementary sets of Q_i . Let $P(A)$ be the set of all subsets of set A .

For any $q \in Q_i$, $G_j(q) \subseteq Q_j$ will be known as the set of processors in Q_j to which processor q will send its results.

A *partnership relation* in Γ is any correspondence $G = (G_i)_{i \in \{0, \dots, m-1\}}$ where

$$\begin{aligned}
 G &: Q \rightarrow \prod_{i=0, \dots, m-1} P(Q_i) \\
 G_j &: Q \rightarrow P(Q_j)
 \end{aligned}$$

In a conventional hypercube, the neighbor or partner of a node in a fixed dimension is unique, while in a partnership relationship G a node may have more than one *partner* in one dimension. This is the reason why functions G_j take their values in $P(Q_j)$ instead of Q_j .

The pair $(\Gamma, (G_i)_{i \in \{0, \dots, m-1\}})$ will be said to be a *neighborhood* if the following conditions are fulfilled:

Exhaustivity: for any $i, j \in \{0, \dots, m-1\}$, any element in Q_j has a partner in Q_i :

$$\forall i, j \in \{0, \dots, m-1\}: \cup_{q \in Q_i} G_j(q) = Q_j \quad (1)$$

This condition guarantees that any processor q in Q_j receives the result of the execution of task T_i performed by the processors in Q_i .

Injectivity: $\forall i, j \in \{0, \dots, m-1\}, i \neq j$

$$\forall q, q' \in Q_j, q \neq q' \text{ it holds } G_i(q) \cap G_i(q') = \emptyset \quad (2)$$

This condition imposes that each processor q in a set Q_i receives the results of task T_j only from one of the processors in Q_j .

If $q \in G_j(q')$, q is said to be a partner of q' in the neighborhood defined by $(\Gamma, (G_i)_{i \in \{0, \dots, m-1\}})$.

A tree or hierarchy of neighborhoods H constitute a *Dynamic Polytope* if it holds that:

1. The root of the tree is the "trivial neighborhood" (Γ, G) where $\Gamma = \{Q\}$ and G is the identity function.
2. If node T is labeled with the neighborhood $(\Gamma^d, (G^d_i)_{i \in \{0, \dots, r-1\}})$ such that $\Gamma^d = \{Q^d_0, \dots, Q^d_{r-1}\}$ is a partition of Q^d , then the children nodes of T are labeled with neighborhoods that partition the sets Q^d_i of Γ^d . Eventually, some of the sets Q^d_i of Γ^d may remain without division (in which case they are leaves in the tree).

The depth of the neighborhood tree will be named *dimension* of H . The nodes at the same level d of the hierarchy H constitute what will be called a *dimension of the Dynamic Polytope*. A dimension is generically designated by the value of its level minus one, and therefore, the first trivial dimension (Γ, G) is dimension -1 .

3.3 Translation Scheme

Fig. 2 presents the expansion of the code in Fig. 1. In general, each time a processor set is divided by the execution of a Division Function, a new dimension (Γ^{dim}, G^{dim}) is created. The creation of the neighborhood (in line 12) can be accomplished in time proportional to the number of tasks demanded (degree). The initial set of processors, Q is divided in two subsets $\Gamma^{dim} = \{Q_0, Q_1\}$ of equal size using constant time. Processors in Q_0 perform the task $pDC(x_0, r_0)$ and those in Q_1 execute $pDC(x_1, r_1)$. A partition Γ maximizing the load balance has to satisfy the equation $|Q_0| / n_0 = |Q_1| / n_1$; where n_i is the number of leaves in the tree of calls produced by the call to $pDC(x_i, r_i)$. In La Laguna C [2] there are some Division Functions having weights, w_i as additional parameters that estimate the quotient $|Q_i| / n_i$. Each of the processors q in Q_0 will hold in variable r_0 the result of $pDC(x_0, r_0)$. At the end of the execution of the parallel tasks, each processor $q \in Q_0$ sends the result r_0 to its partner processors in $G_1(q)$. Since the injectivity conditions of the dynamic polytope holds, only one message is received by each processor (lines 17 and 23 in the code of Fig. 2). On the other hand, the exhaustivity condition guarantees that at the end of line 26 all processors in Q_i get a copy of the result r_{1-i} . The cost of the code in lines 10-26 is dominated by the communication time:

$$D * \max(|r_0| * \max_q(|G_1(q)|), |r_1| * \max_q(|G_0(q)|))$$

where $|r_i|$ represents the size of the results, $|G_i(q)|$ denotes the cardinal of $G_i(q)$ and D is a constant. The time invested in a call to $pDC(x, r)$ obeys the recursive expression:

$$\begin{aligned} \Phi(pDC(x, r), N, g_N, L_N) &= W(\text{divide}(x, x_0, x_1)) + W(\text{PARALLEL}) + \\ &+ \max(\Phi(pDC(x_0, r_0), N_0, g_{N_0}, L_{N_0}), \Phi(pDC(x_1, r_1), N_1, g_{N_1}, L_{N_1})) + \\ &+ g_N * \max(|r_0| * \max_q(|G_1(q)|), |r_1| * \max_q(|G_0(q)|)) + L_N \end{aligned}$$

where $N_i = |Q_i|$ and $N = |Q|$.

```

1  proc pDC(x: Problem; r: Result);
2  begin
3    if trivial(x) then
4      conquer(x, r)
5    else
6      begin
7        divide(x, x0, x1);
8        if (NUMPROCESSORS > 1) then
9          begin
10           save(NUMPROCESSORS, NAME, Gdim(NAME));
11           dim++;
12           compute(Γdim, Gdim);
13           if NAME ∈ Q0 then
14             begin
15               pDC(x0, r0);
16               send r0 to processors in G1dim(NAME);
17               receive r1 from j ∈ Q1 / G0dim(j) = NAME;
18             end;
19           else /* in Q1 */
20             begin
21               pDC(x1, r1);
22               send r1 to processors in G0dim(NAME);
23               receive r0 from j ∈ Q0 / G1dim(j) = NAME;
24             end;
25           dim--;
26           restore(NUMPROCESSORS, NAME, Gdim(NAME));
27           end;
28         else
29           begin
30             pDC(x0, r0);
31             pDC(x1, r1);
32           end;
33         combine(r, r0, r1);
34       end;
35   end;
```

Fig. 2. Translation of the code in Fig. 1; expansion of the PARALLEL call.

4. Computational Results

This section presents the computational results obtained for the implementation under the *common-common* divide and conquer paradigm of some well-known algorithms: the Fast Fourier Transform (FFT) [4] and the Quicksort sorting algorithm [5].

In all cases, the algorithms have been implemented by using *La Laguna C* [2] and they have been executed in three parallel machines: Cray T3E and T3D and IBM SP2. For all the experiments tables with the parallel and sequential times (labels PAR and SEQ respectively) will be presented.

Fig. 3 shows a comparison between the cost of a *La Laguna C* [2] Division Function (label PAR-i) using polytopes and the cost of *MPI_Comm_split* (label MPI-i). The figure represents the time in seconds of successive repetitions of the division functions varying the number of processors. The cost can be observed to be several orders of magnitude greater for the MPI Division Function. In fact, the three curves corresponding to *La Laguna C* Division Function appear overlapped with the horizontal axis.

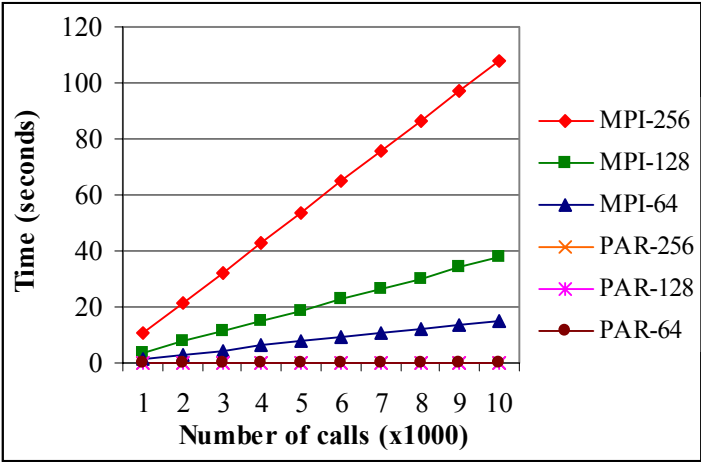


Fig. 3. The cost of *MPI_Comm_split()* vs. *La Laguna C* Division Functions

Table 1. Time in seconds for the sequential and parallel versions of the Fast Fourier Transform algorithm. Size of problem: 256K

P	SP2		Cray T3E		Cray T3D	
	SEQ	PAR	SEQ	PAR	SEQ	PAR
2	4.11	2.13	2.25	1.17	11.51	6.23
4	4.09	1.24	2.25	0.67	11.59	3.67
8	4.06	0.86	2.25	0.43	11.51	2.50
16	4.17	0.63	2.26	0.32	11.59	1.96

Table 1 presents the time in seconds for the FFT algorithm using a 256K complex number vector as input data. The speed-up grows logarithmically when the number of processors increases. The Cray T3E is the architecture that presents a better performance.

Table 2 summarizes the results obtained for the Quicksort algorithm. In this case, the results correspond to the average of ten experiments changing the input data of the problem that were generated according to a uniform random distribution. For this

algorithm, the load balance between the processor subsets is achieved using *La Laguna C* Division Functions that assign processors to the parallel tasks proportionally to the size of the sub-vector to be ordered in each recursive call. The exact structure of the dynamic polytope produced by the Quicksort algorithm depends of the input data distribution.

Table 2. Time in seconds for the sequential and parallel versions of the Quicksort algorithm for an array of 7M integer components

P	Cray T3E		SP2	
	SEQ	PAR	SEQ	PAR
2	8.48	6.26	9.55	7.18
4	8.48	4.07	9.65	4.16
8	8.48	2.84	9.66	2.81
16	8.48	2.15	9.56	2.02

5. Conclusions

This paper presents the NBSP model whose main elements are Division Functions, Collective Operations and the parameters that characterize the inter-processor communications. Our attention has been focused on the efficient implementation of Division Functions applied to *common-common* algorithms. The conditions for the partnership relations have been stated to ensure the correctness of the implementation of any Division Function for a *common-common* problem.

Acknowledgments

We would like to thank the CCCC, CIEMAT and the EPCC for allowing us the access to their computers.

References

1. Valiant, L. G.: *A Bridging Model for Parallel Computation*. Communications of the ACM, Vol. 33(8), (1990) 103-111
2. de Sande, F.: *El Modelo de Computación Colectiva: Una Metodología Eficiente para la Ampliación del Modelo de Librería de Paso de Mensajes con Paralelismo de Datos Anidado*. PhD thesis. Universidad de La Laguna, (1998)
3. Hill, J., McColl, B., Stefanescu, D., Goudreau, M., Lang, K., Rao, B., Suel, T., Tsantilas, Bisseling, R.: *BSPLib: The BSP Programming Library*. Technical Report PRG-TR-29-97, Oxford University Computing Laboratory (1997). www.bsp-worldwide.org
4. Akl, S. G.: *The Design and Analysis of Parallel Algorithms*. Prentice-Hall (1989)
5. Hoare, C. A. R.: *Algorithm 64: Quicksort*. Communications of the ACM Vol. 4, 321 (1961)
6. Preparata, F. P., Shamos, M. I.: *Computational Geometry. An Introduction*. Springer-Verlag, New York (1985)

An MPI Implementation on the Top of the Virtual Interface Architecture^{*}

Massimo Bertozzi, Franco Boselli, Gianni Conte, and Monica Reggiani

Dipartimento di Ingegneria dell'Informazione
Università di Parma, I-43100 Parma, Italy
{bertozzi,boselli,conte,reggiani}@ce.unipr.it

Abstract. This paper describes an implementation of the LAM MPI suite on the top of the Virtual Interface Architecture.

The Virtual Interface Architecture (VIA) is an emerging standard designed by Intel, Microsoft, and Compaq aimed at the reduction of communication latency for cluster of workstations or system area networks.

Thanks to M-VIA, a Linux software module that emulates VIA and provides programmers of VIA APIs, it has been possible to develop an MPI implementation even in absence of a hardware VIA interface. Nonetheless, M-VIA module high optimization as well as VIA protocol simplicity permitted a reduction of latency time with respect to the use of the TCP/IP protocol on Ethernet network interfaces.

1 Introduction

Thanks to the widespreading of NOW prototypes [1], message passing is going to be the most used communication paradigm for parallel machines. Standard programming environment and libraries have been designed to enable the programmers to write fully portable parallel code. Amongst others, the *Message Passing Interface* (MPI) has rapidly received large acceptance because it has been carefully designed to maximize performance on a large variety of systems [8].

Unfortunately, the use of MPI on cluster of workstations generally relies on OS services such as the TCP/IP protocol. This has an obvious disadvantage, mainly the high latency associated to the requirements that TCP enforces in order to work in almost all conditions across highly heterogeneous platforms and WANs. The main advantage is that the programmer is automatically endowed of a truly multi user, reliable programming environment. In addition, most NOW prototypes rely on off-the-shelf network solutions like Ethernet (generally Fast—or, even, Giga—Ethernet), switched Ethernet, ATM, or FDDI. While these solutions provide a good price/performance ratio from the point of view of the bandwidth, they require the operating system intervention for dispatching messages from and to the applications, therefore introducing another latency in the link between application and network.

^{*} The work described in this paper has been carried out under the financial support of the Italian *Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST)* in the framework of the MOSAICO (Design Methodologies and Tools of High Performance Systems for Distributed Applications) Project.

Many solutions to these problems have been proposed, ranging from the development of hardware components dedicated to speedup communications [2, 6, 3] to the implementation of light communication protocols [11, 15, 10]. Nevertheless, no one of the proposed solutions has gained a wide acceptance.

Starting from previous experiences [16], a consortium composed by Intel, Compaq, and Microsoft introduced a new standard for cluster of workstation: the *Virtual Interface Architecture* (VIA), that seems the most promising solution for reducing the communication latency and speeding up the application–network link. The VIA standard, in fact, endorses a direct interface between user applications and network layer without requiring participation of the operating system. The first products based on the VIA standard are now appearing and the proposing consortium seems strong enough to guarantee a wide acceptance of this network standard.

This paper presents an implementation of the LAM MPI suite on the top of the Virtual Interface Architecture. Thanks to M-VIA, a Linux software module that emulates VIA and provides programmers of VIA APIs, it has been possible to develop an MPI implementation even in absence of a hardware VIA implementation.

This paper is organized as follows. Section 2 discusses communication issues and introduces VIA. Section 3 presents the implementation, while section 4 discusses the results and performance of the final product. Section 5 ends the paper with some final remarks.

2 The Communication Architecture

An ideal communication interface/protocol for cluster of workstations or system area networks should feature the following characteristics: low communication latency, high bandwidth, small overhead, and small CPU intervention.

A number of solutions have been proposed to overcome these problems: U-Net [16], Active Messages [17], or Fast Messages [14]. These solutions share the underlying idea of giving to user processes direct access to network resources. Unfortunately, none of the above solutions has reached a wide acceptance. More recently, starting from these experiences, a consortium composed by Microsoft, Compaq and Intel authored a new standard, the Virtual Interface Architecture specifically designed for fast interprocess communication in a cluster or System Area Network environment.

2.1 The Virtual Interface Architecture

The basic concepts that have been used for the VIA design are the following:

- the process has to execute a small number of operations for starting, ending, or managing a communication session;
- these operations have to be as simple as possible;
- I/O operations must not require the use of interrupts;
- context switches as well as copies of data have to be avoided whenever possible;
- concurrency for the communication interface must be managed without operating system intervention.

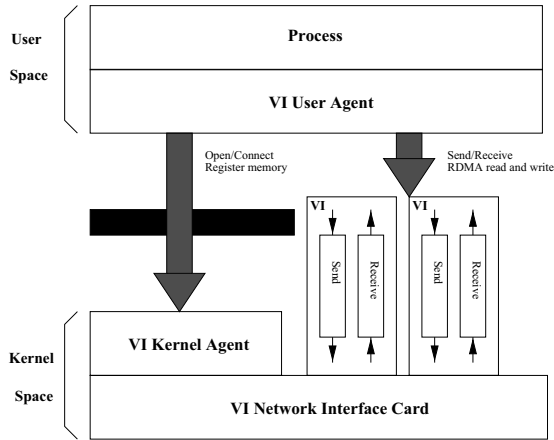


Fig. 1. VIA architectural model.

In the VIA model, the interface driver (usually referred to as the *kernel agent*) performs the setup and manages the resources needed for communications between processes, thus not taking charge of the actual transmission of data. Each process is provided with the exclusive use of a *virtual interface* (VI) for each other process with which it needs to exchange data. The VIs are featured by two queues for sending and receiving data (see figure 1).

VIA provides two types of data transfer facilities: Send/Receive and Remote Direct Memory Access (RDMA). The Send/Receive model follows a standard point-to-point communication model. On both sides, the sender and the receiver must specify the memory regions containing the data to send or where the received data will be placed. In RDMA, the communication initiator specifies both the source buffer and the destination buffer of the data to transfer.

In order to avoid intermediate copies of data, *descriptors* are used: when a process exchanges data with another, it puts a descriptor at the end of the appropriate queue and notifies the VI Network Interface Card writing in a memory mapped register (the *doorbell*). Since now and until the end of the transmission only the interface takes care of the data to be transmitted. The drawback of this mechanism is that the area of memory to be transferred must be *registered* to allow interface access, thus requiring an operating system intervention; this could be generally performed in an initialization phase and not for each transmission.

Similarly, when VIA ends a data transmission the descriptor is completed and the value of the doorbell is changed. The use of these memory mapped registers allows the transfer to proceed without the use of interrupts, that represent a major penalty for modern super-scalar CPUs [9].

Each descriptor is composed by at least two segments. The control segment contains information about the descriptor type, the number of following segments, and the length of data to be transferred. The data segments carry information about local mem-

ory regions and their size. For RDMA operation additional address segments are used containing the virtual address of remote memory region to be transferred.

VIA manages each queue in a FIFO fashion but the completion order of descriptors posted on different VIs is not defined. Synchronization between the user agent and VIA could be obtained both using blocking primitives or polling the head of each queue.

Although the concepts behind the VIA standard are not new, its main advantage is that it has been carefully designed to be independent of operating system and processing architecture, therefore sharing a significant goal with the MPI standard [12] and allowing programmers to write fully portable applications. As a consequence, an MPI implementation with the VIA standard support is straightforward.

2.2 The M-VIA Linux Module

The VIA architecture has been created to be easily integrated in silicon as well as emulated via software [7]. For this reason, a number of research groups have produced a software emulation of VIA that allows a quick port of applications toward the VIA standard even in absence of a hardware VIA interface.

Amongst others, the *National Energy Research Scientific Computing Center* of the *National Laboratories Lawrence Berkeley* developed a VIA software emulation [13] for clusters of Linux boxes and Fast—or Giga—Ethernet interconnections.

The M-VIA module allows development of VIA code and, in addition, features higher communication performance with respect to the TCP/IP protocol thanks to the specific network adapter enhancements and to the simpler VIA communication protocol [4, 5]. On the ParMa² cluster (see section 4) M-VIA features a latency of 61 μ s while TCP/IP latency is 100 μ s.

3 Implementation

3.1 The MPI Suite

There are several MPI implementations that target various parallel architectures and operating systems. Amongst others, the most used and widespread are MPICH from the *Mississippi State University* and *Argonne National Laboratory*, and LAM-MPI from the *Supercomputer Centre of The Ohio State University* and *University of Notre Dame*. Both these implementations support Linux as operating system and cluster of workstation as architecture. In addition, they are multiprotocol and designed to allow an easy integration of new communication layers. In both cases, the source code is freely available and therefore they seem good choices for adding VIA support.

In order to choose one of these implementations, they have been compared on the performance side. A number of benchmarks on a cluster of homogeneous SMP Linux PCs interconnected by Fast Ethernet (see section 4) demonstrated that LAM-MPI is, generally, slightly faster than MPICH [4]; more specifically:

- the peak bandwidth reached using LAM-MPI is 10% larger than the one reached by MPICH and is nearly the maximal bandwidth for Fast Ethernet;
- the latency of LAM-MPI is larger than the MPICH one;

- LAM-MPI is faster when primitives for process synchronization are used;
- in a SMP environment LAM-MPI improves the overall performance through the use of shared-memory, whilst MPICH does not support shared-memory operations.

Moreover, the internal structure of LAM-MPI seemed to us clearer and simpler than the one of MPICH.

3.2 Implementation of a VIA-Based Protocol for LAM/MPI

Several variations on how the sending of a message can interact with the execution of the process are available in the Message Passing Interface standard. Common distinctions are between synchronous or asynchronous sends, and blocking or non-blocking communications. The building blocks to construct MPI communication instructions are the two data transfer facilities available in VIA: Send/Receive and RDMA.

To obtain a successful communication with Send and Receive operations, the receiver must post a descriptor on a Receive Queue of adequate size before the sender's data arrives. In RDMA communication the receiver process is not in charge of any operation during the transfer and no notification is given to the remote node that the request has completed. To achieve a synchronous transfer RDMA with Immediate Data must be used. In fact specifying Immediate Data in the initiator of an RDMA Write request leads to the consumption of a descriptor on the remote end when the data transfer is completed.

The plain use of the communication models provided by VIA, Send/Receive and RDMA, does not allow implementation of MPI based on VIA, due to the large number of communication flavors provided by MPI. Therefore a protocol based on the VIA instructions supporting the whole set of MPI communication features should be conceived.

In order to carry out asynchronous communications it is required to have descriptors in the Receive Queue (RQ) available for future sends. As VIA allows pre-posting of descriptors before the VI connection, the solution is to prearrange a suitable number of descriptors in the Receive Queue. Obviously, in order to execute a number of Send/Receive larger than the pre-posted descriptors a reuse of those already consumed by the Sender is required.

The other critical point is that the memory area where the messages are located must be registered by the Kernel Agent. This operation is computationally expensive so it is impossible to obtain good performance if a new memory area is registered during each communication. Therefore the area where the message will be placed must be registered during the starting phase, before any communication. Of course, it is a waste of memory to register an area equal to the maximum dimension of the messages for each descriptor in the RQ. Another solution is to register a smaller area size for each descriptor and split longer messages. However, this choice results in larger number of communications.

The use of RDMA operations avoids the previous difficulties: as all the communications can use the same RDMA area no memory is wasted due to a higher estimation of memory requirements. Moreover, the use of Immediate Data option in RDMA allows

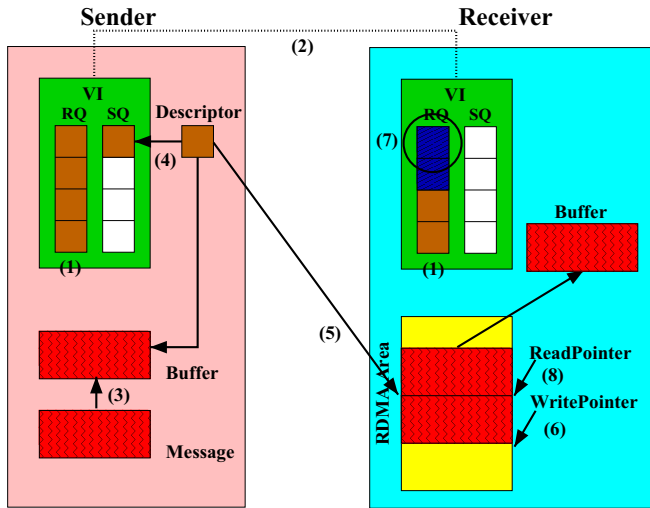


Fig. 2. MPI VIA-based communication protocol.

synchronous transfer. Using RDMA results also in a simpler protocol because only the dimension of the RDMA area must be assessed.

The communication between two MPI processes using RDMA with Immediate Data proceeds as follows. During the starting phase the descriptors are pre-posted in the Receive Queue (1 figure 2), the VIs of the two processes are connected (2) and the RDMA Areas are registered. After exchange of address of the memory regions which will contain the messages, the processes can communicate. To carry out a send, a process must verify whether there is enough free space for the message in the RDMA area of the receiver using the *ReadPointer* and *WritePointer* variables that define the unused area.

When this prerequisite is satisfied the Sender copies the message in the buffer (3) then using the RDMA Write operation a descriptor is inserted in its Send Queue (4) and the message is written in the RDMA Area of the receiver starting at address *WritePointer* (5). Then the number of available descriptors and the variable showing position available for the next send operation in RDMA Area are updated (6).

In the other side in order to execute a Receive, the receiver must verify whether there is a completed descriptor in its Receive Queue (7). When this occurs it reads the message located in the RDMA area at address *ReadPointer*. Then the value of *ReadPointer* is updated (8) and the descriptor is posted again in the RQ.

During the VI connection lifetime the value of the number of descriptors available on the Receive Queues and of the variables *ReadPointer* and *WritePointer* are kept consistent using asynchronous RDMA operations between Sender and Receiver.

The protocol described above has been inserted in the LAM/MPI library. As LAM is composed by an Upper Layer which defines the MPI user functions, and a Lower

Layer performing the actual data transfer on the network only this second layer has been modified in order to use the Virtual Interface Architecture as communication protocol.

4 Results

The implementation of VIA support for LAM-MPI has been developed and tested on the ParMa² cluster of PCs. ParMa² is composed by four dual 450 MHz Pentium PCs with 256 Mbyte RAM each. They are interconnected by a switched Fast Ethernet network based on a CISCO Fast Ethernet switch and full-duplex capable Digital DC21140 Fast—Ethernet network adapters.

The target of this work has been to add the support for VIA to LAM-MPI. In absence of a hardware VIA implementation, a software emulation has been used, therefore performance issues should not matter.

Nonetheless, thanks to the simplicity of the VIA protocol and to the ability of M-VIA in using specific features of supported network adapters, latency of LAM-MPI on VIA is smaller than the one of LAM-MPI on TCP/IP.

In order to evaluate performance, a simple *ping-pong* involving two processes on different nodes of the ParMa² cluster has been used. One process issues an MPI_Send and then waits on an MPI_Recv. The other process receives the message and retransmits it back. Table 1 shows the latencies obtained with LAM-MPI on VIA and on TCP/IP for different message sizes.

Message size (byte)	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384	32768
LAM on M-VIA (μ s)	98	98	98	98	100	104	111	125	151	213	328	470	675	1070	1884	3854
LAM on TCP/IP (μ s)	132	132	132	132	137	140	146	158	183	233	335	486	680	1122	1931	3718

Table 1. Latency for LAM-MPI on VIA or TCP/IP.

For small messages, the latency of LAM-MPI on VIA is nearly one quarter smaller than the one of LAM-MPI on TCP/IP. Anyway, for larger messages the memory copy introduced by our implementation (as explained in section 3.2) becomes significant and reduces the advantages of using VIA. A solution to this drawback is currently under implementation: the communication layer initially checks the message size and then switches on the fastest available protocol.

5 Conclusions

In this paper an implementation of LAM-MPI on top of the Virtual Interface Architecture has been presented.

Thanks to a software emulation of VIA, the porting has been possible even in absence of a hardware interface. Nevertheless, thanks to the simplicity of the VIA communication protocol LAM-MPI on VIA has been proved to reduce latency with respect to the same implementation on the TCP/IP protocol.

The VIA based LAM-MPI implementation is freely downloadable from:
<http://www.ce.unipr.it/pardis/parma2/>.

References

1. G. Bell. 1995 Observations on Supercomputing Alternatives: Did the MPP Bandwagon Lead to a Cul-de-Sac? *Communications of the ACM*, 30(3):325–348, 1996.
2. M. A. Blumrich, K. Li, R. Alpert, C. Dubnicki, E. W. Felten, and J. Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Procs. International Symposium on Computer Architecture '94*, pages 142–153, 1994.
3. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, Jan.–Feb. 1995.
4. F. Boselli. Integrazione della Virtual Interface Architecture (VIA) all'Interno della Libreria di Message Passing LAM/MPI. Master's thesis, Università degli Studi di Parma - Facoltà di Ingegneria, 1999.
5. L. Bougé, J.-F. Méhaut, and R. Namyst. Efficient Communications in Multithreaded Runtime Systems. *Lecture Notes in Computer Science*, 1586:468–482, Feb. 1999.
6. C. Dubnicki, A. Bilas, C. Yuqun, S. N. Damianakis, and L. Kai. Myrinet communication. *IEEE Micro*, 18(1):50–52, Jan.–Feb. 1998.
7. D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A. M. Merrit, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2):66–69, Mar.–Apr. 1998.
8. R. Hempel. The MPI Standard for Message Passing. *Lecture Notes in Computer Science*, 797:247–252, Sept. 1994.
9. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.
10. H. Lu, S. Dworkadas, A. L. Cox, and W. Zwaenepoel. Message Passing Versus Distributed Shared Memory on Networks of Workstations. In *Procs. Supercomputing '95*, 1995.
11. P. Marenzoni, G. Rimassa, M. Vignali, M. Bertozzi, G. Conte, and P. Rossi. An Operating System Support to Low-Overhead Communications in NOW Clusters. *Lecture Notes in Computer Science*, 1199:130–143, Feb. 1997.
12. MPI Forum. MPI A Message Passing Interface Standard. Technical report, University of Tennessee, June 1995.
13. National Energy Research Scientific Computer Center. M-via: A high performance modular via for linux. <http://www.nersc.gov/research/FTG/via/>.
14. S. Pakin, V. Karamcheti, and A. A. Chien. Fast Messages: efficient, portable communication for workstation clusters and MPPs. *IEEE Concurrency*, 5(2):90, Apr. 1997.
15. T. Sterling, D. Savarese, B. Fryxell, K. Olson, and D. J. Becker. Communication Overhead for Space Science Applications on the Beowulf Parallel Workstation. In *Procs. High Performance Distributed Computing - HPDC '95*, Pentagon City, Virginia, USA, 1995.
16. T. von Eicken, A. Basu, V. Buch, and W. Vogels. U-Net: a user-level network interface for parallel and distributed computing. *Operating Systems Review*, 29(5):40–53, Dec. 1995.
17. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. Active Messages: A mechanism for integrated communication and computation. In *Procs. 19th Symp. Computer Architecture*, Gold Coast, Qnd. Australia, May 1992.

MiMPI: A Multithread-Safe Implementation of MPI

F. García, A. Calderón, J. Carretero

Facultad de Informática, Universidad Politécnica de Madrid (UPM)
Campus de Montegancedo, Boadilla del Monte, 28660 Madrid, Spain
`fgarcia@fi.upm.es`

Abstract. In this paper, we present a new thread-safe implementation of MPI, called MiMPI, that allows efficient development of thread-safe parallel programs using the message-passing standard MPI. MiMPI uses multithread operations to increase the performance of collective communications. The paper describes the main design goals of MiMPI and the current implementation. Finally we present some performance results, obtained on an IBM SP2, comparing the performance with others MPI implementations.

Keywords: Parallel programming, message-passing, MPI, threads.

1 Introduction

MPI [MPI95] defines a standard interface for the *message-passing model* of parallel computation. MPI was intended to facilitate widespread portability of programs among diverse parallel architectures. With this aim, the primary goals of the MPI specification are: efficiency, portability, and functionality.

Threads are useful to improve application performance. A program with only one thread of control must wait each time it requests a service from the operating system. Using more than one thread lets a process overlap processing with one or more I/O requests.

Using threads in conjunction with message passing and the use of a multithread implementation of MPI can be extremely convenient, for several reasons:

- Threads provide a natural implementation of nonblocking communication operations. A thread can be created to do a blocking receive operation. This operation blocks only the thread and not the process. The same applies to sends.
- Threads can increase the efficiency of the implementation of collective operations.
- Threads provide a natural way to implement operations required for the shared-memory operations. A separate set of threads, one for each process, can handle the data management functions on behalf of a process while a *main* thread in each process performs the main computation.
- Threads are becoming the parallel programming model for symmetric multiprocessing shared-memory machines.

- Threads are specially important for client/servers applications. Server programs in client/servers applications may get multiple requests from independent clients simultaneously.
- Threads can improve performance by helping to reduce communication latency.

MPI specification defines a thread-safe semantic. Thread safety means that multiple threads can be executing message-passing library calls without interfering with one another. MiMPI is a multithread implementation of MPI, that uses threads to increase the performance of some operations and provides an implementation that can be used in multithread applications.

Currently MiMPI implements a subset of all MPI functions, and have been used successfully in *ParFiSys* [Garcia98], a parallel file system developed at the UPM. MiMPI is available for cluster of workstations and IBM SP2 platforms.

The rest of this paper is organized as follows. Section 2 surveys related work about MPI implementations. In section 3, we discuss our thread-safe implementation of MPI. Section 4 provides some experimental results comparing MiMPI with other implementations. The experiments have been made on an IBM SP2. Finally, section 5 summarizes our conclusions.

2 Related Work

MPI [MPI95] [Gropp95] (*message-passing interface*) is a message-passing application programmer interface, together with protocol and semantic specifications for how its features must behave in any implementation.

MPI includes point to point message passing and collective operations. A collective operation must be performed by all the processes in a computation. MPI supports two kinds of collective operations: data movement operations and collective computation operations. Others features included in MPI are: virtual topologies, debugging and profiling, communication modes, and support for heterogeneous networks.

In 1997 appears MPI-2. MPI-2 contains clarifications and corrections to the initial standard and describes additions to this standard. These include miscellaneous topics, process creation and management, one-sided communications, extended collective operations, external interfaces, I/O, and additional language bindings.

Currently, there are many implementations of MPI, with both free available and vendor-supplied implementations, but few implementations provide a fully thread-safe semantic. Some of these systems are as follows:

- LAM [Burns94] is available from the Ohio Supercomputer Center and runs on heterogeneous networks of Sun, DEC, SGI, IBM, and HP workstations.
- CHIMP-MPI [Alasdair94] is available from the Edinburgh Parallel Computing Center and runs on Sun, SGI, DEC, IBM, and HP workstations, the Meiko Computing Surface machines. This implementation is based on CHIMP [Clarke94].

- MPICH [Gropp96] is an implementation developed at Argonne National Laboratory and the Mississippi State University. This implementation is available for several platforms, Sun, SGI, RS6000, HP, DEC and Alpha workstations and multicomputers as the IBM SP2, Meiko CS-2 and Ncube.
- Unify [Vaughan95], available from Mississippi State University, layers MPI on a version of PVM that has been modified to support contexts and static groups. Unify allows MPI and PVM calls in the same program.
- IBM's MPI for the SP2. IBM uses this implementation as the *native* language for programming SP2 systems, replacing the non-standard MPL. IBM's MPI provide a fully thread-safe semantic. With these implementations users can develop multithread applications in conjunction with MPI functions.
- MPI-LITE [Bhargava97] provides a portable kernel for thread creation, termination, and scheduling. This implementation can be used in multithread applications. However, this model is very strict due to uses user-level threads, each thread executes a copy of the given MPI program, and the total number of threads in the program are specified as inputs to the MPI-LITE program. This implementation does not support dynamic creation and termination of threads.

3 MiMPI Implementation

MiMPI (*multithread implementation of MPI*) is a new implementation of MPI. The main design goals of MiMPI are:

- To support a thread-safe semantic, suitable for develop multithread applications using MPI.
- To use threads in the MiMPI implementation. The main goal is to increase the performance of the collective and nonblocking operations. In collective functions we create one thread for each individual operation (see figure 1).
- To be easily portable to different platforms.
- To provide a graphical user interface for MPI. The primary function of this interface is to start MPI programs and to visualize what is going on inside a MPI application that use MiMPI.

MiMPI architecture is clearly divides in three levels (see figure 2):

- Basic services that hides hardware differences from upper levels of the operating systems, to help make MiMPI portable.
- Level of lightweight communications, called XMP.
- The third level implements the MPI interface.

The first level hides hardware differences from upper levels of the operating systems. The current implementation use POSIX services and TCP/IP protocols using UNIX sockets.

XMP can be used to implement others message-passing model, as PVM. This communication micro-kernel provides the following interface:

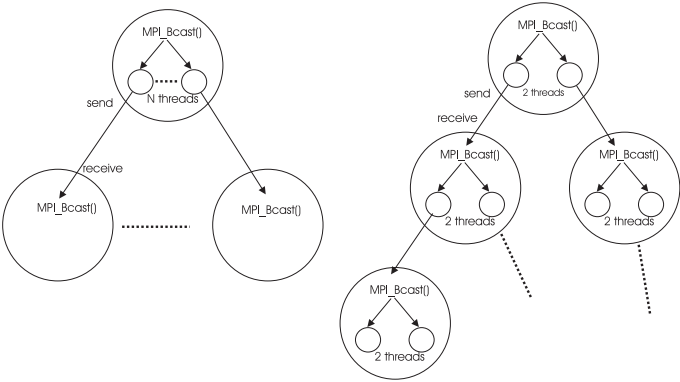


Fig. 1. Implementation of collective operations

- `XMP_Init()`
- `XMP_Finalize()`
- `XMP_Request(struct request *)`

`XMP_Init()` and `XMP_Finalize()` functions are used to start and finalize the system. `XMP_Request()` is employed for the basic message-passing operations: *send* and *receive*.

The last level provides the MPI implementation and uses the former levels.

MiMPI includes a graphical user interface to start the execution of MPI programs. Currently this interface is available for cluster of workstations. In the future we want to extend this interface to visualize MiMPI applications.

The current implementation of MiMPI runs on UNIX and Linux cluster of workstations, and IBM SP2 multicomputers.

4 Evaluation

This section describes the performance experiments designed to evaluate the MiMPI implementation. These benchmarks, based on the code described in [Miguel96], are described in the next sections.

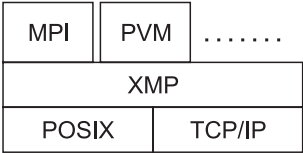


Fig. 2. Architecture of MiMPI

4.1 Point to point communication

This benchmark characterizes point to point communications using MPI. Two process, 0 and 1, engage in a sort of ping-pong, with process 0 in charge of the measurements. This process reads the value of the clock before invoking a `MPI_Send` operation and then blocks in a `MPI_Receive` (meanwhile, process 1 performs the symmetric operations). Once the latter operation finishes at process 0, the clock is read again. Thus, the delay of a two-message interchange (one is each direction) has been measured. The latency is computed as one half of this time and the achieved throughput is also computed, considering the latency and the message size. These operations are done 1000 times to average results. We have considered average values, because they are more representative of the performance the user can obtain from a machine.

We have also modified this benchmark. The main goal for the new benchmark is to characterizes point to point communication between several threads (see figure 3). Process 0 and 1 create n threads. The threads in process 0 engage in a sort of ping-pong with threads in Process 1. In this test we obtain the time need to finish the ping-pong and the time to create and destroy the n threads. The throughput is computed considering data interchanged in the former time.

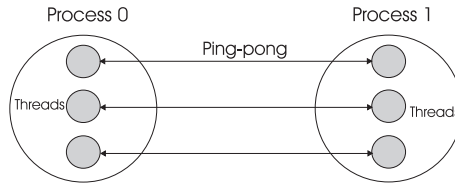


Fig. 3. Ping-pong multithread

4.2 Collective communications

Several MPI collective operations have also been measured. In this paper we present the performance of the *broadcast* (`MPI_Bcast`), *gather* (`MPI_Gather`), and *scatter* (`MPI_Scatter`) operations.

The broadcast test involves 8 processes performing a broadcast from process 0 (the root of the operation) to all of them. The test performs 1000 iterations, again to average values. In each iteration, all process perform a broadcast operation and then a barrier-synchronize (`MPI_Barrier`). To compute the broadcast delay, the root measures the time from the moment the broadcast starts until the time the barrier finishes. The figures indicate the aggregated throughput obtained, including the time to perform the barrier operation.

The gather and scatter tests are like the previous one, but using `MPI_Gather` and `MPI_Scatter`. Figures consider the aggregated throughput obtained for several message lengths to receive or to send respectively.

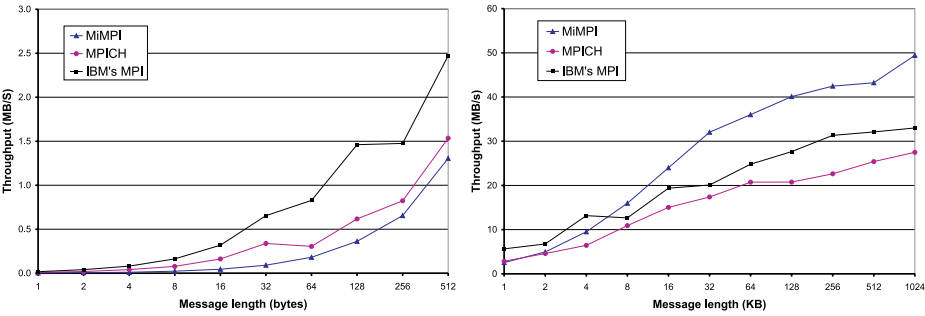


Fig. 4. Throughput for point to point communication

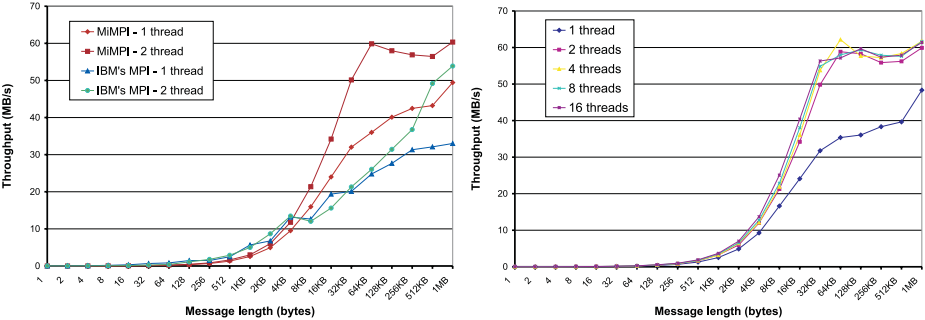


Fig. 5. Throughput for ping-pong multithread (2 threads). 16 threads in MiMPI

4.3 Results

The former benchmarks have been executed on an IBM SP2 machine. Each node has a 66 MHz POWER2 RISC System/6000 processor with 256 MB of memory, running AIX 4.2 operating system version. In the SP2 we have compared the performance of MiMPI with the IBM's MPI and MPICH implementations. MiMPI and IBM's MPI provides a thread-safe implementation of MPI, but MPICH not. All test uses the IBM's high performance switch.

Figure 4 compares the throughput obtained in the point to point communication. MiMPI provides better results for longer message, due to worse latency obtained using TCP/IP protocol for short messages.

Figure 5 depicts the results obtained for the point to point multithread communication. This test has been used with MiMPI and IBM's MPI, because these implementations are both thread-safe. However, this test fail using IBM's MPI for more than 2 threads. So therefore, figure of the left shows results for only 2 threads. The figure of the rigth shows results using several threads. As is shown by this figure the aggregated throughput is maintained when the number of threads is increased. This demonstrate that threads can reduce communication latency.

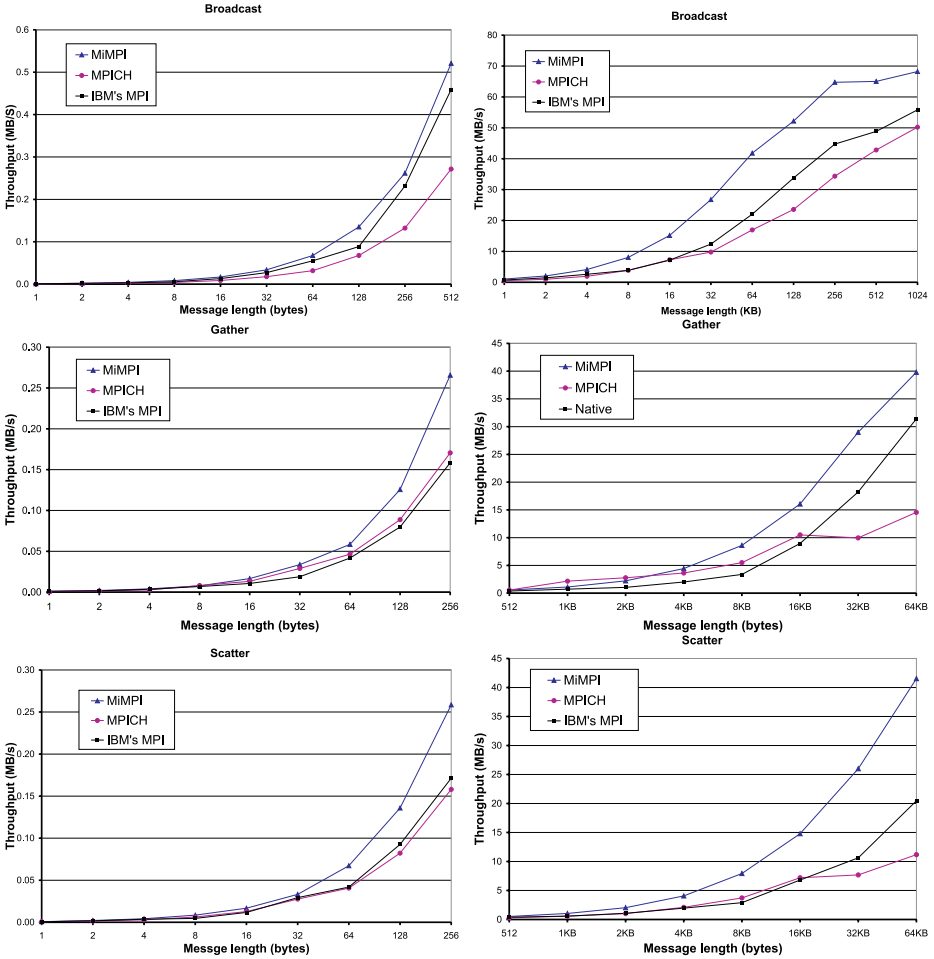


Fig. 6. Throughput for broadcast, gather and scatter

Finally, figure 6 shows the throughput obtained for the collective operations. As can be observed, MiMPI performs better even for short messages, mainly due to the use of multithread operations in all collective functions (see figure 1). This demonstrates the convenience of using threads to implement collective operations.

5 Conclusions and Future Work

The paper has presented a new implementation of MPI. MiMPI is a multithread implementation of MPI that provided a fully thread-safe semantic, suitable for multithread applications using MPI. As we can see in the evaluation performed, MiMPI increases the performance of MPI collective functions using multithread operations.

Further work is going on to implement all MPI functions, to optimize MiMPI to reduce the latency for short messages, and to finalize the graphical user interface to visualize what is going on inside a MiMPI application.

Acknowledgments. We want to express our grateful acknowledgment to the CESCA institution for giving us access to their IBM SP2 machine.

References

- [Alasdair94] R. Alasdair, A. Bruce, J.G. Mills, and Smith A.G. CHIMP-MPI user guide. Technical Report EPCC-KTP-CHIMP-V2-USER 1.2, Edinburgh Parallel Computing Centre, 1994.
- [Bhargava97] P. Bhargava. MPI-LITE User Manual, Release 1.1. Technical report, Parallel Computing Lab, University of California, Los Angeles, CA 90095, April 1997.
- [Burns94] G. Burns, R. Daoud, and J. Vaigl. LAM: An open cluster environment for MPI. In *Proceedings of Supercomputing Symposium '94*, pages 379–286. In John W. Ross, editor, 1994.
- [Clarke94] J. Clarke, A. Fletcher, M. Trewin, A. Bruce, A. Smith, and R. Chapple. Reuse, Portability and Parallel Libraries. In *Proceedings of IFIP WG10.3 – Programming Environments for Massively Parallel Distributed Systems*, 1994.
- [MPI95] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard*. 1995. www.mpi-forum.org/docs/mpi-11.ps
- [MPI97] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*. 1997. www.mpi-forum.org/docs/mpi-20.ps
- [Garcia98] F. García, J. Carretero, F. Perez, and P. de Miguel. Evaluating the *ParFiSys* Cache Coherence Protocol on an IBM SP2. Technical Report FIM/104.1/datsi/98, Facultad de Inform'atica, UPM, Campus de Montegancedo, 28660 Madrid, Spain, January 1998.
- [Gropp96] W. Gropp, E. Lusk, N Doss, and A. Skjellum. A High Performance, Portable Implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, (22):789–828, 1996.
- [Gropp95] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI, Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 1995.
- [Langendoen97] K. Langendoen, R. Bhoedjang, and H. Bal. Models for Asynchronous Message Handling. *IEEE Concurrency*, (2):28–38, 1997.
- [Miguel96] J. Miguel, R. Beivide, A. Arruabarrena, and J.A. Gregorio. Assessing the Performance of the New IBM SP2 Communication Subsystem. In *VII Jornadas de Paralelismo*, pages 115–130, September 1986.
- [Vaughan95] P. L. Vaughan, A. Skjellum, D. S. Reese, and F. Chen Cheng. Migrating from PVM to MPI, part I: The Unify System. In *Fifth Symposium on the Frontiers of Massively Parallel Computation*. IEEE Computer Society Technical Committee on Computer Architecture, IEEE Computer Society Press, 1995.

Building MPI for Multi-Programming Systems Using Implicit Information

Frederick C. Wong¹, Andrea C. Arpaci-Dusseau², and David E. Culler¹

¹ Computer Science Division, University of California, Berkeley
{fredwong, culler}@CS.Berkeley.EDU

² Computer Systems Laboratory, Stanford University
dusseau@CS.Stanford.EDU

Abstract. With the growing importance of fast system area networks in the parallel community, it is becoming common for message passing programs to run in multi-programming environments. Competing sequential and parallel jobs can distort the global coordination of communicating processes. In this paper, we describe our implementation of MPI using implicit information for global co-scheduling. Our results show that MPI program performance is, indeed, sensitive to local scheduling variations. Further, the integration of implicit co-scheduling with the MPI runtime system achieves robust performance in a multi-programming environment, without compromising performance in dedicated use.

1 Introduction

With the emergence of fast system area networks and low-overhead communication interfaces [6], it is becoming common for parallel MPI programs to run in cluster environments that offer both high performance communication and multi-programming. Even if only one parallel program is run at a time, these systems utilize independent operating systems and may schedule sequential processes interleaved with the parallel program. The core question addressed in this paper is how to design the MPI runtime system so that realistic applications have good performance that is robust to multi-programming.

Several studies have shown that shared address space programs tend to be very sensitive to scheduling and in many cases only perform well when co-scheduled [1]. Fortunately, this work has also shown an interesting way for programs to co-ordinate their scheduling implicitly by making simple observations and reacting by either spinning or sleeping [2].

The first question to answer is whether or not MPI program performance is sensitive to multi-programming. It is commonly believed that message passing should tolerate scheduling variations, because the programming model is inherently loosely coupled and programs typically send large messages infrequently. It is used, after all, in distributed systems and PVM-style environments. However, local scheduling variations may cause communication events to become *out-of-sync* and impact the global

schedule [1].

In this paper, we show that MPI performance is, indeed, sensitive to scheduling; the common intuition is misplaced. Two or three programs running together may take 10 times longer than running each in sequence; competing with sequential applications has a similar slowdown. This result is demonstrated on the NAS Parallel Benchmarks [3] using a fast, robust MPI layer that we have developed over Active Messages [6] on a large high-speed cluster. We then show that simple techniques for implicit co-scheduling can be integrated into the MPI runtime library to make application performance very robust to multi-programming with little loss of dedicated performance.

This paper is organized as follows. We briefly describe our experimental environment and our initial MPI implementation in Section 2. In Section 3, we examine the sensitivity of message passing programs to multi-programming and show the influence of global uncoordination on application execution time. Section 4 describes our solution to this problem and gives a detailed description of our MPI implementation using implicit co-scheduling. Application performance results are discussed in Section 5.

2 Background

This section briefly describes our experimental environment and our initial implementation of the MPI Standard that serves as a basis for our study of sensitivity to multi-programming.

2.1 Experimental Environment

The measurements in this paper are performed on the U. C. Berkeley NOW cluster, which contains 105 UltraSPARC I model 170 workstations connected with 16-port Myrinet [4] switches. Each UltraSPARC has 512 KB of unified L2 cache and 128 MB of main memory. Each of the nodes is running Solaris 2.6, Active Messages v5.6 firmware, and GLUnix [7], a parallel execution tool used to start the processes across the cluster.

2.2 MPI-AM

Our MPI implementation is based on the MPICH [8] (v1.0.12) reference implementation by realizing the Abstract Device Interface through Active Message operations. This approach achieves good performance and yet is portable across Active Message platforms.

Active Messages. Active Messages [6] (AM) is a widely used low-level communication abstraction that closely resembles the network transactions that underlie modern parallel programming models. AM constitute *request* and *response* transactions which form restricted remote procedure calls. Our implementation of the Active Messages

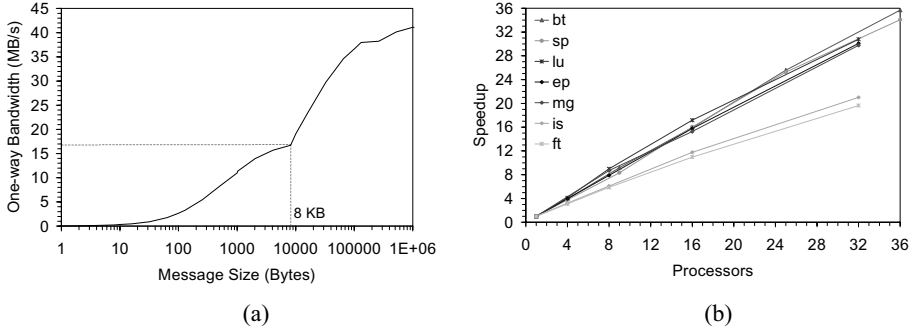


Fig. 1. These figures show the micro-benchmark and application performance of MPI-AM in a dedicated environment. Figure *a* shows the *one-way bandwidth* of MPI-AM with *message sizes* up to 1 MB. Figure *b* shows the *speedup* of the NAS benchmarks (v2.2) on up to 36 processors

API [9] supports a multi-user, multi-programming environment, and yet provides efficient user-level communication access.

Implementation. Our current implementation of MPI uses the *eager* protocol to transport messages in the abstract device. Each MPI message is transferred by a medium AM request (up to 8 KB) with communication group and rank, tag, message size, and other ADI specific information passed as request arguments. Messages larger than 8 KB are fragmented and transferred by multiple Active Messages. Instead of specifying the destination buffer, the sender transfers all message fragments to the destination process, which dynamically reorders and copies the message fragments into the appropriate buffer. A temporary buffer is allocated if the corresponding receive has not been posted. An MPI message is considered delivered when all associated AM requests are handled at the receiving node.

Performance in Dedicated Environment. Figure 1a shows the one-way bandwidth of MPI-AM using Dongarra's echo test [5]. The one-way bandwidth is calculated with the reciprocal of the one-way message latency, which is half of the average round-trip time. The start-up cost of MPI-AM is 19 μ s (3 μ s above that of the raw AM performance) and the maximum bandwidth achievable is 41 MB/sec. The kink at 8 KB shows the performance of a single medium request. The increase in bandwidth with message sizes larger than 8 KB is due to streaming multiple medium AM requests.

Figure 1b shows the speedup of the NAS benchmarks on class A data sizes on up to 36 dedicated processors in the cluster. Except FT and IS, for which the performance is limited by aggregate bandwidth, the benchmarks obtain near perfect speedup. Indeed, for a few benchmarks, speedup is slightly super-linear for certain machine sizes due to cache effects. To evaluate the impact of multi-program scheduling, we chose three applications (LU, FT, and MG) representing a broad range of communication and computation patterns that we might encounter in real workloads.

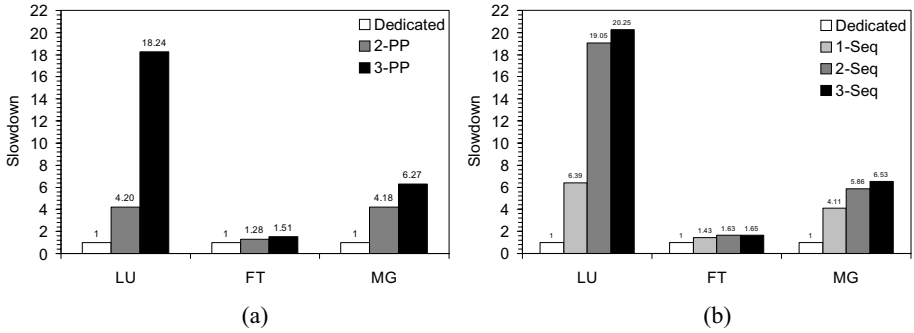


Fig. 2. These figures show the *slowdown* of the NAS benchmarks in a multi-programming environment. Figure *a* shows the *slowdown* of *LU*, *FT*, and *MG* when one copy (*Dedicated*), two copies (*2-PP*) and three copies (*3-PP*) of the same benchmark are running on the same cluster. Figure *b* shows the *slowdown* of the same benchmarks competing with one copy (*1-Seq*), two copies (*2-Seq*) and three copies (*3-Seq*) of a sequential job

3 Sensitivity to Multi-programming

In a multi-programming environment, a local operating system scheduler is responsible for multiplexing processes onto the physical resources. The execution time of a sequential process is inversely proportional to the percentage of CPU time allocated, plus possible context switching overheads. Parallel applications spend additional communication and waiting time in the message passing library. For a parallel application running in a non-dedicated environment, the waiting time may increase if the processes are not scheduled simultaneously. In this section, we examine the sensitivity of parallel applications to multi-programming. In particular, we investigate the performance of the NAS benchmarks when multiple copies of parallel jobs are competing against each other, and when multiple sequential jobs are competing with a parallel job.

Figure 2a shows the slowdown of our three benchmarks when multiple copies of the same program are running. The rest of our experiments are performed on a set of 16 workstations. Slowdown is calculated by dividing the running time of the program in a multi-programming environment by the execution time of the program run sequentially in a dedicated environment. The slowdown of a workload with two jobs is equal to one if the workload runs twice as long as a single job in the dedicated environment. LU has the most significant drop in performance, with a slowdown of 18.2 when three copies of LU are competing for execution. MG and FT have a slowdown of 6.3 and 1.5 respectively. This shows that the performance of the NAS benchmarks is noticeably worse when they are not co-scheduled.

Figure 2b shows the effect on the NAS benchmarks of competing with sequential processes. The benchmark LU has a slowdown of 20.2, FT has a slowdown of 1.7 and MG has a slowdown of 6.5, when three copies of the sequential job are run. As when competing with parallel jobs, the performance of the benchmarks drops significantly when they are actively sharing with sequential jobs.

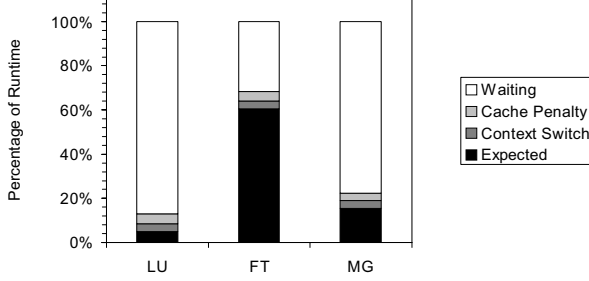


Fig. 3. This figure shows the execution time breakdown of the NAS benchmarks when three copies of a sequential process running

To further understand the performance impact of a multi-programming environment on parallel jobs, we profile the execution time of each benchmark. Figure 3 shows the execution time breakdown of the NAS benchmarks sharing with three copies of a sequential job. The *expected time* is the execution time of the benchmark in a dedicated environment. The *context switch time* is the overhead of context switching to other processes. The *cache penalty* is the extra time spent in the memory hierarchy due to cache misses caused by other processes. The *waiting time* is the additional time spent by processes waiting on communication events in the multi-programming environment.

Although the context switching cost and cache penalty are substantial, they constitute less than 10 percent of the execution time. The excessive slowdown of LU and MG is explained by a tremendous increase in waiting on communication events. For example, only 5 percent of the LU execution time is spent performing useful work while 85 percent of the time is spent on spin-waiting on message sends and receives.

As shown in these figures, parallel jobs are highly sensitive to global co-scheduling. The traditional solution to this problem in MPPs is explicit gang scheduling by the operating system. This is unattractive in general-purpose clusters and mixed workloads. In the following sections, we present an elegant solution obtained by modifying our implementation of MPI to use local information to achieve global co-scheduling.

4 Incorporating Implicit Co-scheduling

The idea of implicit co-scheduling is to use simple observations of communication events to keep coordinated processes running, and to block the execution of un-coordinated processes so as to release resources for other processes. If the round-trip time of a message is significantly higher than that expected in a dedicated environment, the sending process can infer that the receiving process is currently not scheduled. Therefore, by relinquishing the processor of the sending process, other local processes can proceed. On the other hand, a timely message response means the sender is probably scheduled currently, and consequently, the receiving process should remain scheduled.

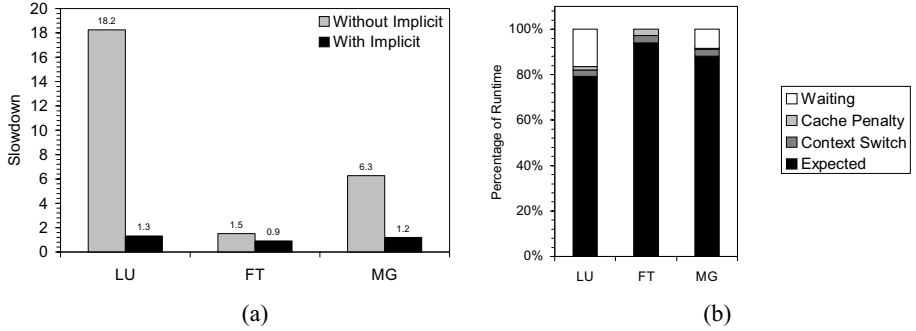


Fig. 4. Figure *a* compares the *slowdown* of the NAS benchmarks with and without implicit co-scheduling on three copies of the same benchmark. Figure *b* shows the *execution time breakdown* of the benchmarks when three copies of a sequential process are time-shared with each of the benchmarks using implicit co-scheduling

Two-phase spin-blocking is a mechanism that embodies these ideas. It consists of an active spinning phase and a block-on-message phase. In the active spinning phase, the sender actively polls the network for a reply for a fixed amount of time. In the block-on-message phase, the caller blocks until an incoming message is received. In order to keep active processes co-scheduled, the amount of time the sender needs to spin wait has to be at least the sum of the round-trip time plus the cost of a process context switch. On the other hand, the receiver should wait for the time of a one-way message latency under perfect co-scheduling.

We utilize these ideas and modify our MPI-AM runtime library to incorporate the two-phase spin-block mechanism wherever spin-waiting may occur. The first two places are trivial; two-phase spin-block is used when the sender is waiting for a reply from the receiver to confirm the delivery of the message (`MPID_Complete_send`), and when the receiver is waiting for the delivery of a message (`MPID_Complete_recv`).

Spin-waiting can also occur when posting a receive (`MPID_Post_recv`). Every MPI receive contains a *receive handle* that carries the receive information. When posting a receive by the application, the abstract device needs to ensure that the network device has not received the corresponding message by checking the *unexpected receive queue*. If no match is found, the *receive handle* is then posted to the *expected receive queue*. Otherwise, the message has been received by the network device and an *unexpected handle* is returned. Since large messages are fragmented, it is possible that the message is in the middle of transmission when the receiving process retrieves the *unexpected handle*. Our implementation spin waits until the entire message is buffered. Thus, the two-phase spin-block mechanism is needed to relinquish the processor if the sender is not delivering the message fragments fast enough.

Finally, we need to ensure that every layer in the message passing system does not spin wait. In our Active Messages implementation, an AM request operation might be blocked due to running out of flow control credits. In order to avoid the spin wait in the AM layer, MPI-AM keeps a outstanding requests counter. Two-phase spin-block is used whenever the counter reaches the pre-defined flow control limit of the AM layer.

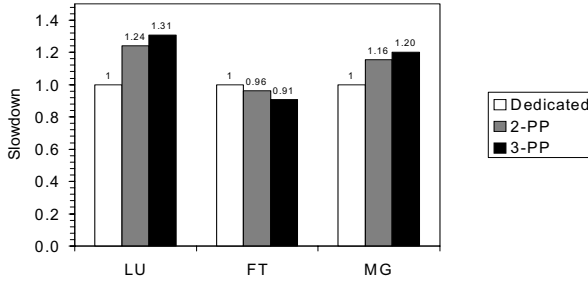


Fig. 5. This figure shows the slowdown of the NAS benchmarks when one copy (*Dedicated*), two copies (*2-PP*), and three copies (*3-PP*) of the same benchmark are run

5 Results

The extra complexity introduced by implicit co-scheduling to the MPI library only increases the one-way latency of small message by $1\ \mu s$ on the echo test, whereas the execution time of a single copy of the NAS benchmarks is essentially unchanged.

Figure 4a compares the slowdown of running three copies of the NAS benchmarks with and without implicit co-scheduling. LU has the most significant improvement, from a slowdown of 18.2 down to 1.3. Figure 4b shows the execution time breakdown of the benchmarks when running with three copies of a sequential process. In LU the waiting time is reduced from 9900 seconds (Figure 3) to 160 seconds, in MG it is reduced from 180 seconds to 3 seconds and in FT, it is almost completely eliminated. The two-phase spin-block mechanism effectively reduces the spin-waiting time in the benchmarks without substantially increasing the context-switch overhead.

Figure 5 repeats the study of Figure 2a using implicit co-scheduling to demonstrate the scalability with different workload sizes. Both LU and MG experience a moderate increase in slowdown when multiple copies are run. This increase occurs because both application perform very frequent fine-grained communications.

FT, on the other hand, experiences a speedup when more copies are run. Because FT sends relatively large messages, its performance is limited by the aggregate bandwidth of the network and processes often spin wait for the messages to drain in and out of the network. In a dedicated environment, FT spends as much as 25 percent of its execution time spin waiting in communication events. When multiple copies of FT are implicitly co-scheduled, a waiting process eventually blocks, allowing other processes to continue. The total execution time of all processes is therefore reduced by interleaving the computation and communication across competing processes. Speedup is achieved when the benefit of interleaving outweighs the cost of multi-programming.

6 Conclusion

Previous studies have shown that implicit co-scheduling is useful for building parallel

applications using a fine-grain shared memory programming model. Our study indicates that loosely coupled message passing programs are highly sensitive to multi-programming as well, even on computationally intensive, bulk synchronous programs. Application slowdown caused by global uncoordination can be as high as 20 times. Without coordination, processes may waste up to 85 percent of their execution time spin-waiting for communication.

In this paper, we have presented an implementation of the MPI Standard for a distributed multi-programming environment. By leveraging implicit information for global co-scheduling, we effectively reduce the communication waiting time in the message passing applications caused by communication uncoordination. The performance of message passing applications is improved by as much as a factor of 10, reducing the application slowdown to 1.5 in the worst case studied. In one case, implicit co-scheduling helps a coarse grain message passing application yield better performance by interleaving communication with computation across parallel processes.

Acknowledgments

We would like to thank Shirley Chiu, Brent Chun, Richard Martin, Alan Mainwaring, and Remzi Arpaci-Dusseau for their many helpful comments and discussions on this work. This research has been supported in part by the DARPA (F30602-95-C0014), the NSF (CDA 94-01156), the DOE (ASCI 2DJB705), and the California MICRO.

Reference

1. R. Arpaci, A. Dusseau, A. Vahdat, L. Liu, T. Anderson, D. Patterson: The Interaction of Parallel and Sequential Workloads on a Network of Workstations. In *Proceedings of ACM Joint Intr. Conf. on Measurement and Modeling of Computer Systems*, pp. 267-278, May 1995.
2. A. Arpaci-Dusseau, D. Culler, A. Mainwaring: Scheduling with Implicit Information in Distributed Systems. *ACM SIGMETRICS'98/ PERFORMANCE'98*.
3. D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga: The NAS Parallel Benchmarks. *Intr. J. of Supercomputer Applications*. 5(3):66-73, 1991.
4. N. Boden, D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su: Myrinet - A Gigabit-per-Second Local-Area Network. *IEEE Micro*, 15(1):29-38, Feb. 1995.
5. J. Dongarra and T. Dunnigan: Message Passing Performance of Various Computers. University of Tennessee Technical Report CS-95-299, May 1995.
6. T. von Eicken, D. Culler, S. Goldstein, and K. Schauer: Active Messages: a Mechanism for Integrated Communication and Computation. In *Proc. of the 19th ISCA*, 1992, pp.256-266.
7. D. Ghormley, D. Petrou, S. Rodrigues, A. Vahdat, and T. Anderson: GLUnix: A Global Layer Unix for a Network of Workstations. *Software Practice and Experience*, 1989.
8. W. Gropp, E. Lusk, N. Doss, and A. Skjellum: A High-Performance, Portable Implementation of the (MPI) Message Passing Interface Standard. *Parallel Computing* 22(6):789-828, Sept. 1996.
9. A. Mainwaring: Active Message Application Programming Interface and Communication Subsystem Organization. University of California at Berkeley, T.R. UCB CSD-96-918, 1996.

The design for a high performance MPI implementation on the Myrinet network*

Loic Prylli¹, Bernard Tourancheau², and Roland Westrelin²

¹ LIP and project CNRS-INRIA REMAP, ENS-Lyon, 69634 Lyon

² RHDAC and project CNRS-INRIA REMAP, ISTIL UCB-Lyon 69622 Villeurbanne
Bernard.Tourancheau@inria.fr; <http://www-bip.univ-lyon1.fr>

Abstract. We present our MPI-BIP implementation, designed for Myrinet networks, and based on MPICH. By using our “Basic Interface for Parallelism: BIP” software layer, we obtain in this implementation of the MPI protocols results close to the peak hardware performance of the high speed Myrinet network. We present the protocols we used to implement the MPI semantics, and the overall design of the implementation. We, then, present benchmarks and application results to show that this design leads to parallel multicomputer-like throughput and latency on a cluster of PC workstations.

1 Introduction

In the last decade, researchers tried to use COWs (Cluster Of Workstations) as parallel computers. These clusters are typically connected by Ethernet networks and are often programmed with communication libraries like PVM (Parallel Virtual Machine [6]), or MPI over IP (Internet Protocol). There is two bottlenecks in these solutions that can restrict application programmers to coarse grain parallelism: either the network hardware (ethernet or fast ethernet) may not be fast enough, or the communication software (and this is particularly true for all IP-based systems) may have too much overhead.

It is now possible to build a platform with efficient hardware and software components. The price performance ratio of COWs depends on:

The processing power commodity components (like Intel Pentium or DEC alpha processors) now deliver competitive performance.

The network speed Massively Parallel Processor technologies (SCI and Myrinet) are available to interconnect workstations.

The operating system free software operating systems (like Linux) provide the flexibility and performance needed.

The communication libraries

This article introduce our works towards a high performance implementation of MPI for Myrinet-based clusters using the BIP protocols. We, first, released an MPI implementation on top of BIP in 1997 [15] using the MPICH Channel Interface. A re-implementation of MPI-BIP was done in 1998 (released in

* This work was supported partly by a Myricom grant and the Région Rhône-Alpes.

November) based on the work done at Myricom on the MPI-GM[1] implementation. The paper describes the design of the last implementation, most of it is shared between MPI-GM and MPI-BIP.

The first part of this paper introduces previous works in this domain and BIP. Then we present the design of MPI-BIP, our adaptation layer protocol for MPICH, followed by some performance results.

2 Low-level layers

Methodologies for the design of protocol stacks changed with the new high performance networks. The bottleneck now can often be the interface between the network and the host or the communication system if badly designed, especially in regards to latency. Several research teams proposed new approaches:

Active Messages (AM) [14] use an RPC-like (Remote Procedure Call) mechanism. Each message begins with the address of a handler. It is executed by the receiver to integrate the data in the ongoing computation. A MPI implementation on top of the AM is available [3].

Fast messages (FM) [11] use the same RPC-like system. They include a heavy flow control protocol. The designers of FM provide an implementation of MPI [9].

PM [8] is a fast communication layer for the myrinet network. MPI was ported to PM [10].

GM [1] is the native API for the myrinet hardware developed by myricom. MPI is also available on top of GM.

Virtual Interface Architecture (VIA) [5] Computer industry leaders (Microsoft, Intel and Compaq) proposed a new software architecture for an efficient access to the network hardware from the user applications. VIA borrows a lot to U-Net [2]. At the time of writing, no free implementation of MPI is available for VIA.

2.1 BIP (Basic Interface for Parallelism)

BIP [12] is a low level layer for the Myrinet network developed by our team in Lyon. The goal of this project is to provide an efficient access to the hardware and to allow zero memory copy communications. BIP only implements a very raw link level flow control. It is not meant to be use directly by the parallel application programmer as it supplies very few functionalities. Instead, higher layers are expected to provide a development environnement with a good functionality/performance ratio.

The API of BIP is a classical message passing interface. BIP provides both blocking and non-blocking communications. Communications are as reliable as the network, errors are detected and in-order delivery is guaranteed. BIP is composed of a user library, a kernel module and a Network Interface Card (NIC) program. The key points of the implementation are:

A user level access to the network, avoiding system calls and memory copies implied by the classical design, becomes a key issue: the bandwidth of the

network (160 MB/s in our case and 133 MB/s for the I/O bus) is the same order of magnitude than the speed of a memory copy.

The long messages follow a rendez-vous semantic: the send is guaranteed to complete only if a corresponding receive has been posted because of the limited capacity of the network. Messages are split into packets and the communication steps are pipelined (see [13]).

The small messages, because initializations and handshakes between the host and NIC program are more expensive than a memory copy for small messages, are directly written in the network board memory on the sending side and, copied in a queue in main memory on the receiving side. The size of this queue is static, it is up to the application to guarantee that no overflow occurs. On the receiving side, a memory copy puts the message in the user buffer platform.

The communication latency of BIP is about $5.0 \mu\text{s}$, the maximal bandwidth is 126 MB/s (95% of the PCI maximum which is the bottleneck in our case). Half the maximum bandwidth is reached with a message size of 4KB ($= N_{1/2}$).

3 The internals of MPI-BIP

MPICH is organized in layers and designed to ease the porting to a new target. The figure 2 presents our view of the MPICH framework, and at what level we plugged the MPI-BIP specific part, different ports choose different strategies depending on what communication system they use.

There is an internal intermediate API inside MPICH called the “Channel Interface” often used for new ports. This API requires to implement a non blocking send for the control messages (small messages used to exchange informations between the MPI processes, as explained below) and the possibility to queue a high number of asynchronous requests. BIP does not fulfill these requirements. That is why we did not plug it at the “Channel Interface” level¹. We implemented our network specific layer at a non-documented interface level, that we called the “Protocol Interface”, because this API allows to specify custom protocols for the different kinds of MPI messages.

Each MPI messages of the application, is implemented with one or several messages of the underlying communication system (BIP in our case). We distinguish two logical kind of BIP messages that we use to implement MPI: the “control” messages, and the “large data” messages.

3.1 “Control” messages

The control messages are sent directly to the network when the MPI internals need to exchange some control information. If the receiver does not expect a control message, it is stored in the queue of BIP. This queue is static (i.e. of constant size). Even if the size of a control message varies (depending on the size of the data that are included or on the type of informations exchanged by

¹ The original MPI-BIP implementation worked around this with some tricks.

the upper layers), an upper bound can be determined. Hence, the maximum number of messages that the queue can hold is known. To guarantee that we do not send a number of messages greater than this maximum, a flow control algorithm is used (see 3.5).

There are mainly four kind of control messages:

- small application message (encapsulated in a control message),
- request of transmission of a large message,
- acknowledgement, sent when the receiver is ready for a requested transmission,
- credits (used for flow control).

3.2 “Large data” messages

Large messages are used to transfer the user-data for MPI messages larger than a fixed threshold. Such messages cannot get into a statically allocated queue. As described below, a preliminary handshake with “control messages” is necessary, to ensure that the receiver is ready before the user data is actually sent.

3.3 The internal protocols

MPICH originally implements three different protocols on top of the “Channel Interface” (see figure 2) described in[7]:

- short protocol: data is added to the payload,
- eager protocol: a control message is sent, and the payload is sent separately just after,
- rendez-vous protocol: after sending a control message, the sender must wait an acknowledgement before sending the payload.

For MPI-BIP, we needed to partly modify these protocols. First the eager protocol is not usable with BIP (because it does not allow the destination to prepare itself before the emission of the large message starts at the other end), instead we replaced it with a three-way protocol that will either allocate a temporary buffer on the receiving side if the user did not post any matching receive yet, or will directly receive into the user-buffer. Then the rendez-vous protocol has been extended to allow the payload to be sent in several fragments separated by acknowledgements to workaround the message size limitations of BIP.

The correspondence between MPI communications and those three transmissions is done with the following rules:

- MPI Synchronous sends always use the rendez-vous protocol,
- MPI Buffered sends are managed by a generic layer of MPICH and transformed into Standard sends,
- MPI Standard and Ready sends use either the short protocol, the three-way protocol or the rendez-vous protocol depending on their size.

These rules determine completely what kind of BIP messages will be needed to implement a MPI message of the application.

3.4 Request FIFOS

The protocols above rely on the ability to do any number of non-blocking network operations (so no deadlock is possible in their executions). We need some additional mechanisms on top of BIP to ensure this non-blocking capability on the sending side for two reasons:

- BIP allows only to queue a limited number of asynchronous sends, when the maximum is reached we would need to wait before posting a new send.
- The flow control mechanism for control messages described in the next section may prevent us to send immediately to a given destination.

When these situations occur, we put any new send request in a software FIFO, it will be transformed into a BIP send later (upon receiving flow control credits, or on completion of previous sends). This software FIFO actually allows to present to the upper layers of MPI-BIP a virtual communication systems supporting an unlimited number of non-blocking sends.

We have to deal with a similar resource limitation problem on reception. When receiving a request for a large message, one of the requirements before sending the acknowledgement, is to first post the BIP receive request. As only a small number of simultaneous receives can be posted with BIP (one for each tag), we need a FIFO containing receive requests that need to be delayed. If a request is put into this FIFO, the corresponding acknowledgement will only be sent at the time the request goes out of the FIFO.

3.5 The flow control algorithm for the control messages

In order to avoid overflow of the fixed-size receive queues of BIP, a credit-based algorithm is used to manage the control messages.

A machine i manages two credit accounts for each other machine j : $C(i \rightarrow j)$ which is the number of messages i can send to j at a given time, $B(i \leftarrow j)$ and the number of consumed messages from j for which i have not yet sent back corresponding credits.

Each time a control message is send from i to j : $C(i \rightarrow j)$ is decremented, the value of $B(i \leftarrow j)$ is put in a dedicated header of the message, after which $B(i \leftarrow j)$ is reset to zero.

On j , upon reception of a control message from i , the credit value in the header is added to $C(j \rightarrow i)$, and $B(j \leftarrow i)$ is incremented.

Let N be the number of credits for each peer of processes, (at all times we have $C(i \rightarrow j) + B(j \rightarrow i) \leq N$), on machine i , when $B(i \leftarrow j)$ reaches $N - 1$, we need to force the emission of a “credits only” control message to j . In usual applications, such dedicated messages will be very rare, credits will go back in the header of other control messages.

Note that we must always reserve the last credit for the eventuality of a “credits only” control message, so N must be at least 2 (other “control message” goes into the previously mentioned FIFO if $C(i \rightarrow j) \leq 1$).

4 Performances measurements

Our LHPC² experimentation platform is made with 8 nodes, each with an Intel Pentium Pro 200 MHz, 64Mbytes of RAM and a 440FX chipsets, and a myrinet board with LANai 4.1 and 256Kbytes of memory.

4.1 Point to point communications

We ran point to point measurements for communications between two Myrinet connected Pentium-Pros running Linux. The performance measurement method uses the median of 100 experiments. They consist in round trip communications: the message is sent, the receiver returns it only when it is fully received. The one way timing is obtained by dividing the round trip timing by 2.

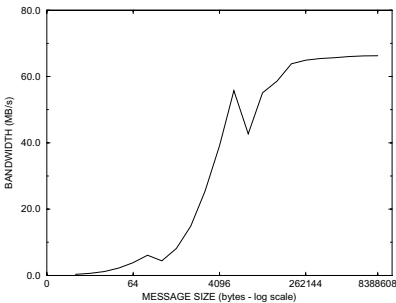


Fig. 1. Point to point communication performance: bandwidth vs message size in logarithmic scale

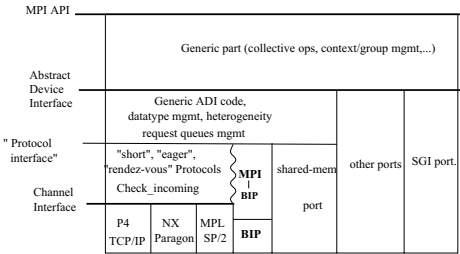


Fig. 2. The architecture of MPI-BIP

	Latency (μs)	Bandwidth (MBytes/s)	$N_{1/2}$
Intel Paragon (MPI)[7]	40	70	≈ 7000
IBM SP-2 (MPI)[4]	35	35	3262
T3D (MPI)[7]	21	100	≈ 7000
SGI Power Challenge (MPI)[7]	47	55	≈ 5000
Myrinet/Ppro200 (BIP)	5	126	4096
Myrinet/Ppro200 (MPI-BIP) 1st implementation from 1997	9	125	8192
Myrinet/Ppro200 (MPI-BIP) (with 10kb fragments)	9.5	66	3072
Myrinet/Ppro200 (MPI-BIP) (with 100kb fragments)	9.5	110	8000

Table 1. Comparison of communication performance with some parallel architectures.

Figure 1 gives the bandwidth of MPI-BIP. Three artifacts are clearly visible. They match the 3 strategies used: small messages are sent directly; medium messages are sent using the three-way protocol of MPICH; big messages are split into several fragments (10Kbytes for this experiment) using the rendez-vous protocol, (note that the frontier between three-way and rendez-vous cannot be seen in the case of a ping-pong experiment).

² Laboratoire pour les Hautes Performances en Calcul: a cooperation structure between ENS-Lyon, r gion Rh nes Alpes, CNRS, INRIA and Matra Syst me Information.

These results are very good and our software puts the Myrinet network in the range of the parallel machines as shown in Table 1. Note that although it was the fastest, the original implementation from 1997 is now obsolete, because of some MPI conformance problems.

4.2 Running the NAS benchmarks

The NAS parallel benchmarks (<http://science.nas.nasa.gov/Software/NPB/>) are a set of parallel programs designed to compare the performance of supercomputers. Each program tries to test a given aspect of parallel computation that could be found in real applications. These benchmarks are provided both as single processor versions and parallel codes using MPI. We selected 3 different benchmarks (LU, IS and SP) and compiled them with MPI-BIP. They are then run on 1, 4 and 8 nodes of our Myrinet cluster described previously. The table 2 gives our measurements and comparisons with several parallel computers. Data for parallel computers come from the NAS web site.

	MPI-BIP on PPro 200		IBM SP (66/WN)		Cray T3E-900		SGI Origin 2000-195		Sun Enterprise 4000	
	Mop/s	Speedup	Mop/s	Speedup	Mop/s	Speedup	Mop/s	Speedup	Mop/s	Speedup
IS @ 4 proc	2.44	4.5	2.2	3.1	3.2	N/A	2.1	3.8	1.6	3.8
IS @ 8 proc	2.11	8.8	2.0	5.6	3.8	N/A	2.4	8.6	1.5	6.9
LU @ 4 proc	23.68	6	59.0	3.7	67.6	N/A	96.8	N/A	36.9	4.1
LU @ 8 proc	22.73	11.6	57.2	7.1	66.4	N/A	103.3	N/A	37.3	8.4
SP @ 4 proc	10.10	3.4	42.1	3.6	43.0	N/A	60.3	N/A	25.0	3.7

Table 2. Performance for NAS Benchmarks on various platforms.

The speedup values summarized in the table 2 show the very good performance of MPI-BIP and Myrinet. Super-linear speedups in our case can be explained by the large cache size and the super-scalar architecture of the PPro. The behavior of MPI-BIP is excellent both for small messages and large messages. However, when a lot of computational power is needed (SP), a gap appears between traditional parallel machines and our cluster: the weak floating-point unit of the PPro 200 shows its limits!

5 Future works

Our work proves that the performance gap between workstation clusters and parallel machines is nearly filled in. However, even if parallel machines are still more powerful, COW based on inexpensive hardware has an unbeatable performance price ratio.

Our future work on MPI-BIP will investigate the increase of the performance by implementing some of the flow control at the BIP level in the Network Interface Card program. We would especially like to improve the throughput of MPI and enable a better communication/computation overlap and also get rid of the performance artifacts.

We are trying to generalize our approach with a new MPICH architecture that should ease the porting of MPI on new high speed network architectures. Furthermore, we are studying the modifications needed at the BIP level to support efficiently the MPI 2 functionalities. The software has been downloaded by more than 110 different teams worldwide and researcher are using it for large applications.

References

1. Myricom gm. <http://www.myri.com:80/GM/>.
2. Anindya Basu, Vineet Buch, Werner Vogels, and Thorsten von Eicken. U-Net: A user-level network interface for parallel and distributed computing. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.
3. Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, and Thorsten von Eicken. Low-latency communication on the IBM RISC system/6000 SP. *ACM/IEEE Supercomputing '96*, November 1996.
4. Jack Dongarra and Tom Dunigan. Message-passing performance of various computers. Technical Report CS-95-299, University of Tennessee, July 1995.
5. Dave Dunning and Greg Regnier. The Virtual Interface Architecture. In *Hot Interconnects V*, Stanford, USA, August 1997.
6. Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, 1994.
7. W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, September 1996.
8. Yutaka Ishikawa Hiroshi Tezuka, Atsushi Hori. Pm:a high-performance communication library for multi-user parallel environments. Technical Report TR-96015, RWC, 1996. <http://www.rwcp.or.jp/lab/pdslab/papers.html>.
9. M. Lauria and A. Chien. MPI-FM: High performance MPI on workstation clusters. *Journal of Parallel and Distributed Computing*, February 1997.
10. Francis O'Carroll, Atsushi Hori, Hiroshi Tezuka, and Yutaka Ishikawa. The design and implementation of zero copy MPI using commodity hardware with a high performance network. *ACM SIGARCH ICS'98*, pages 243–250, July 1998.
11. S. Pakin, V. Karamcheti, and A. Chien. Fast messages (FM): Efficient, portable communication for workstation clusters and massively-parallel processors. *IEEE Concurrency*, 1997.
12. Loïc Prylli and Bernard Tourancheau. BIP: a new protocol designed for high performance networking on myrinet. *Workshop PC-NOW, IPPS/SPDP98*, 1998.
13. Loïc Prylli, Bernard Tourancheau, and Roland Westrelin. Modeling of a high speed network to maximize throughput performance: the experience of BIP over myrinet. *Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, 1998.
14. T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Int'l Symp. on Computer Architecture*, Gold Coast, Australia, may 1992.
15. Roland Westrelin. Réseaux haut débit et calcul parallèle: étude de myrinet. Master's thesis, ENS Lyon, Université Lyon 1, INSA Lyon, 1997. <http://lhpc.univ-lyon1.fr/PUBLICATIONS/pub/RWdea.ps.gz>.

Implementing MPI's One-Sided Communications for WMPI

Fernando Elson Mourão and João Gabriel Silva

CISUC/Departamento de Eng. Informática, Universidade de Coimbra,
Pólo II, 3030 Coimbra, Portugal
`elson@dsg.dei.uc.pt`, `jgabriel@dei.uc.pt`

Abstract. One-sided Communications is one of the extensions to MPI set out in the MPI-2 standard. We present here a thread-based implementation of One-sided Communications written for WMPI, an existing Windows implementation of MPI written at the Universidade de Coimbra. This is a major step towards WMPI incorporating the MPI-2 standard, with the further benefit of contributing to the thread safety of WMPI. We discuss the main design decisions associated with the implementation and consider further research work required in this area to improve both the existing implementation and to assess other implementations of One-sided Communications.

1 Introduction

MPI is the de facto standard for message passing, and its acceptance is so wide that the demand for new features increases rapidly. So the MPI Forum released the MPI-2 standard[1] in June 1997. This paper describes the implementation of one of the most important new chapters in the standard, the One-Sided Communications (OSC) chapter. This implementation is an extension to an existing Windows implementation of MPI and represents the first step towards MPI-2 compliance.

This paper is laid out as follows. Firstly section 2 gives background information placing this implementation of OSC into context. In section 3 the implementation is discussed, including major design decisions and performance issues. Following this, section 4 suggests directions for further research and work, and finally section 5 concludes the paper.

This work was partially supported by the portuguese Ministério da Ciência e Tecnologia and the European Union through the R&D Unit 326/94 (CISUC), the project ESPRIT IV 23516 named WinPar and the project PRAXIS XXI 2/2.1/TIT/1625/95 named ParQuantum.

2 Background

The implementation of OSC discussed here was done over an existing Windows implementation of MPI, the Windows Message Passing Interface (WMPI) [2, 3].

WMPI is now in the process of being extended to meet the requirements of the MPI-2 standard [1]. The implementation of OSC forms part of this work. Below we give brief details of WMPI, the OSC chapter in the MPI2 standard and an overview of OSC.

2.1 WMPI

WMPI was the first implementation of MPI for computers running the Windows operating system. This implementation was originally based on MPICH [4–7] but it has been tuned and recently Mark Baker showed[8] that WMPI was the fastest Windows implementation freely available. The idea behind WMPI is to take advantage of the evergrowing number of Windows based machines and that purpose has been achieved.

2.2 MPI 2

MPI-2.0, as stated by the MPI Forum, is a set of extensions to the MPI-1.1 standard. These extensions include a defined way of running MPI processes, C++ bindings and thread compliance. However its main new features are discussed in its four main chapters:

- Process Creation and Management is a first simple approach to allow MPI applications to launch more MPI processes during runtime, a feature mostly needed in networks of workstations (NOWs) and clusters of PCs (COPs).
- Parallel I/O concerns the parallel and distributed environments but is out-with the message passing scope.
- Extended Collective Operations are a true extension to the existing collective operations, but also an extension to allow collective operations to cope with Process Creation and Management.
- One-Sided Communications are asynchronous communications that allow one process to specify both the sending and receiving parameters for the message being transferred, hence the name "one-sided". This also means that the remote process involved does not have to explicitly call any MPI function to send or receive the message.

As is evident from their names, only one, Extended Collective Operations, refers to pure message passing. The reasons for this seem to be related to the fact that MPI is being used so widely that there is a need to cover other areas aside from pure message passing. Moreover, nowadays the use of NOWs and COPs for parallel computing is a reality, which seems to have driven the MPI Forum to take into consideration the needs of these types of machines in presenting chapters such as Process Creation and Management.

2.3 One-Sided Communications

The MPI Forum also names OSC function calls as remote memory access (RMA) calls. There is a set of synchronisation functions to control the access to remote

processes' memory, and a set of RMA functions to retrieve and put data into a remote process's memory.

For RMAs to be issued a group of processes have to call an initialisation function, `MPI_Win_Create`. There each process states the amount of memory that is available for remote access, as well as giving a pointer to that space. When the RMAs are finished the processes call `MPI_Win_free` to release the memory and close remote accesses.

Between these two calls any number of synchronisation and RMA calls can be issued. The synchronisation calls open what the standard refers to as **epochs**. Epochs can be **access epochs** or **exposure epochs**. If a process A is issuing RMAs to a process B then process A must have an access epoch open and process B must have an exposure epoch open. There are three types of synchronisation calls that can be used:

Fence (`MPI_Win_fence`) is a global (to the group of processes that initially called `MPI_Win_Create`) synchronisation call which opens both exposure and access epochs in all the processes.

Start/Post (`MPI_Win_start`, `MPI_Win_complete`, `MPI_Win_post`, `MPI_Win_wait`) are two pairs of synchronisation calls that open and close an access epoch and a corresponding exposure epoch on a group of processes. The accessing processes call start and complete, which respectively open and close an access epoch, while the targeted processes have to call post and wait to respectively open and close an exposure epoch.

Lock (`MPI_Win_lock`, `MPI_Win_unlock`) is a one-to-one synchronisation call that opens an access epoch at the calling process and an exposure epoch at the given target. The exposure epoch is opened without the target process having to call any synchronisation call or even being aware of its memory being accessed.

The RMA calls are:

- `MPI_Get` to read data to remote processes.
- `MPI_Put` to write data to remote processes.
- `MPI_Accumulate` to write data to remote processes but using an operation over the existing data.

To avoid repeating the standard refer to the MPI 2.0 Standard document for further details.

3 Implementation

This implementation is a first prototype and improvements are expected. In particular since many of the implementation options were tightly restricted. As well as the standard's requisites there was already a fully running implementation of MPI which had not been planned to satisfy the needs of OSC. Thus some options taken were driven by the fact that it would not pay off to undertake certain changes to the existing code. The most relevant ones relate to the asynchronous

agent to handle the requests, and the issue of datatypes handling. This section discusses the most important implementation options taken and the reasons for them.

3.1 Synchronisation Model

The standard states that OSC follows a loose synchronisation model. For that the synchronisation function calls should only block when strictly necessary. However the standard allows an implementation to block on all synchronisation calls if desired. The implementation discussed here behaves as follows:

- The fence call blocks when it is closing an epoch.
- The start call blocks if any of the processes in the group has not yet closed a previous epoch from the calling process.
- The complete call only blocks if there are RMAs requests waiting for a reply.
- The post call does not block.
- The wait call blocks until all processes in the accessing group call complete.
- The lock call only blocks if the target process is the local and there is a lock being held already. Locks to remote processes do not block under any circumstance.
- The unlock call blocks until all the issued RMAs receive a reply and until the lock epoch is closed at the target process.

This behaviour is more complex to implement than blocking all calls, but copes better with network latency in COPs. Moreover if all calls were to block then the whole purpose and advantage of loose asynchronous communications would be lost.

3.2 To Thread or Not to Thread

In NOWs and COPs there is no native support for RMAs, so an asynchronous agent is required to handle requests. This can be achieved either by using specific hardware or by implementing it with software. There are several ways of implementing it depending on the system it is being implemented for. The two approaches we considered for OSC were:

- All MPI calls check for asynchronous OSC requests.
- Use a separate software agent such as a thread or a dedicated process.

The first option would require all MPI calls to check if a request for RMAs had been issued and if so the request would be dealt with. This option requires less dramatic changes to the existing code, but it is easy to see that if a targeted process does few calls to MPI the performance is affected. Scalability is poor and the method is prone to process starvation and deadlocks. It also might cause delay on simple MPI calls if a reasonable number of OSC requests are on hold and have to be processed.

The second option could be implemented using a process or using threads. If a process were to be implemented then a large amount of data would have to be shared between this process and the MPI processes. Thus interprocess communication mechanisms such as semaphores and shared memory would have to be used intensively. These mechanisms, along with context switching between processes, are very expensive in terms of performance. Thus threads were considered to be a better option.

Having decided to use threads a second decision was required: to use only one thread per process or one thread per window. In the first case one thread would serve all windows of a given MPI process. In the second case each window which the process creates has a thread associated with it. Our conclusions were that one thread per process could easily become a bottleneck, is obviously more complex and in certain ways defeats the purpose of using threads.

In the implementation scheme used, a thread is created in the `MPI_Win_create` call each time a new window is created. The thread is destroyed by the `MPI_Win_free` call.

3.3 Datatypes

The datatypes handling functions are not required to be global operations. The WMPI implementation relies on this fact to make datatype handling calls local.

Considering that the internal data representation can differ from machine to machine, sending or receiving data using OSC becomes an issue. When using regular MPI send and receive calls this is not a problem because when data is received the local datatype is used to un-marshal the data. However when using OSC the datatype is unknown at the receiving end. Both datatypes (send and receive) are given as parameters at the process issuing the RMA. Thus the controller thread has no information about what datatype to use to marshal or un-marshal the data.

The solutions found to this problem are:

- Pack and send the needed datatype information with each request.
- Change datatype handling function calls to become global operations.
- Use datatype caching to improve performance over the first option presented here.

The solution implemented was the first of these, due to the fact that the other two required further research and effort which was beyond the project's scope. A more detailed discussion on the last two options is presented in section 4.

3.4 Performance

As stated before performance was not the priority for this project. Although some benchmarking was planned it could not be performed.

The planned benchmarking was to be done using third party benchmarking applications. However no suitable applications were found. This can be explained

by the fact that there are few implementations of OSC and those that do exist are not in the public domain.

A first formal analysis suggests that the results are likely to be below that expected for a high performance library. The most obvious reason for lower performance is the datatype information sent with each RMA request. A less obvious reason is that the actual RMA requests are sent and processed individually. However an algorithm that takes advantage of the loose synchronisation model could improve the performance of the current implementation. The following section describes some of these as a subject of further research.

4 Further research

This section highlights areas where further research and work on the OSC implementation is needed. While this list is not exhaustive we believe it covers the major issues.

4.1 Datatypes Handling

As discussed in section 3.3 handling datatypes proved to be a matter of concern in implementing OSC. Datatypes are local to MPI processes but in OSC MPI processes need to know about datatypes belonging to other processes.

At the moment each time an RMA is issued the required datatype information is sent with the request. However we suggest two approaches to improve this solution: global IDs for datatypes and caching of datatype information.

Global IDs: If the datatypes were identified by a global ID then the problem no longer exists, as all processes will have the required information. However to achieve this all current MPI datatypes handling calls would have to become collective. There are three main disadvantages to this option:

- It does not follow the standard.
- It adversely affects performance.
- Under dynamic process creation the propagation of global IDs and datatypes' information to the new processes has to be done, i.e. we still have the same problem.

Cache of Datatypes Information: If the datatype information currently sent with each RMA request were cached by the controller thread, then subsequent RMAs would not need to send the information again. This approach exploits locality of reference in long running high performance applications. Additionally it does not require changes to the current WMPI implementation and is an extension to the OSC implementation presented here.

4.2 RMA Grouping

There is potential to improve the handling of RMA requests. Instead of issuing RMA calls individually, grouping them could cope better with latency and low bandwidths. Further research is needed to find the optimal grouping scheme, or to develop an adaptive algorithm to suit the needs of an application at a given time. For instance, one issue to be considered is the tradeoff between the size of the message and the time required to process the number of RMAs in the message.

4.3 Benchmarking

Benchmarks are fundamental to high performance libraries. They are not only a way of assessing improvements but more importantly to spot areas that need improvement. For OSC we consider that benchmarking should concentrate on the synchronisation calls as these are the ones that require more processing and data checking.

5 Conclusion

In this paper we have described the implementation of One-Sided Communications for WMPI. The first results are satisfying, but the lack of proper benchmarking applications restricted the project work. Further work is required in the area, in particular to develop benchmarks, assess the effects of grouping RMA accesses and to improve datatype handling.

References

1. Message Passing Interface Forum, "MPI-2: Extensions to the Message-Passing Interface", June 1997.
2. Marinho, José, Silva, João Gabriel: WMPI - Message Passing Interface for Win32 Clusters, in Proc. of 5th European PVM/MPI Users' Group Meeting, pp. 113-120, September 1998.
3. José Marinho: Realização Prática da Norma MPI para Redes de Computadores Pessoais, MSc thesis, August 1996, Universidade de Coimbra
4. William Gropp, Ewing Lusk, "MPICH Working Note: Creating a new MPICH device using the Channel interface - DRAFT", ANL/MCS-TM-000, Argonne National Laboratory, Mathematics and Computer Science Division
5. William Gropp, Ewing Lusk, "MPICH ADI Implementation Reference Manual - DRAFT", ANL-000, Argonne National Laboratory, August 23, 1995
6. Ralph Butler, Ewing Lusk, "User's Guide to the p4 Parallel Programming System", Argonne National Laboratory, Technical Report TM-ANL-92/17, October 1992, Revised April 1994
7. W. Gropp, E. Lusk, N. Doss, and A. Skjellum: "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard", Pre-print MCS-P567-0296, July 1996.

8. Mark Baker, "MPI on NT: The Current Status and Performance of the Available environments", in Proc. of 5th European PVM/MPI Users' Group Meeting, pp. 63-73, September 1998.

A Parallel Genetic Programming Tool Based on PVM

F. Fernández ¹, J.M. Sánchez ¹, M. Tomassini. ², and J.A. Gómez ¹

¹ Departamento de Informática. Escuela Politécnica. Universidad de Extremadura. Cáceres
fferveg@alcazaba.unex.es, sanperez@unex.es, jangomez@unex.es

² Institut d'Informatique. Université de Lausanne
mtomassi@iissun4.unil.ch

Abstract. This paper presents a software package suited for investigating Parallel Genetic Programming (PGP) utilizing Parallel Virtual Machine (PVM) language as means of communicating distributed populations. We show the usefulness of PVM by means of an example developed with this software tool. The example has been run on several processors in a parallel way.

1 Introduction

Genetic Programming (GP) is a machine learning methodology developed by John Koza in the early 1980s [1], capable of finding solutions for difficult problems. GP is based on the Darwin's Natural Selection theory [2] and attempts to solve a problem with an unknown solution which is represented as a program.

Basically the evolutionary process can be regarded as follows:

1. An stochastic process seeds the initial population with a lot of programs.
2. A fitness function is used to evaluate each individual, i.e. a program, of the population.
3. The best individuals are selected, crossed over and mutated to create a new population, which will be the next generation.
4. Steps above are repeated evaluating individuals and producing new generations until a good enough individual is found.

Although the evolutionary process has an evident parallel nature, because all individuals develop and fight for survival, almost all GP implementations developed to date are sequential. The algorithm is implemented by means of a sequential programming language and is executed in a sequential computer, which evaluates step after step each population, applying basically crossover and mutation in order to create new generations.

In this paper we show a parallel algorithm, utilized to construct a software tool capable of evolving simultaneously several populations. Each population consists of individuals, which are programs. Each program is represented as a tree. In this work we demonstrate how PVM can be used for communicating populations.

In the following section we introduce the methodology applied, later we detail the developed software. In Section 4 we show an example with the results. Finally in the last section we express the conclusion of this work.

2 Methodology

In order to develop a standard software package we need to choose the programming language we will use. Although GP was first developed using LISP language, C and C++ languages have used later. Nowadays there are several available software packages for working with Genetic Algorithms, GA, and GP, many of them written in C or C++ programming language [3]. Furthermore we must consider that our system must be parallel and will be run on as many machines as possible, including heterogeneous ones. PVM is perfectly adapted to this idea. It is written in standard C allowing its use in several computer systems. PVM also allows sending and receiving information among several programs running simultaneously in different machines.

In short, we decided to use C and specially C++ as implementation language. At the same time we looked for an available and proved tool for GP, written in C++. We wanted to add the necessary code for converting it into a new tool, which allows parallel run of populations. Finally we chose GPC++0.5.2 [4]

At this stage it's necessary to underline that there are two basic differences between sequential and parallel processing of a problem when GP is applied. On the one hand, we have the difference due to the algorithmic point of view. When a certain problem is solved using GP, exchanging individuals among several populations has been proved useful, when these populations work independently looking for solutions for the same problem. This exchange of individuals seems to speed up the convergence to the required solution [5]. On the other hand, we see that in sequential implementations we have only one population, whose individuals are evaluated sequentially, but in the parallel version as many individuals as population are evaluated simultaneously. It is clear that the parallel algorithm is different from the sequential one.

Moreover, in parallel processing each population can be assigned to a different processor. Each processor evaluates the individuals of the population sequentially, but on the whole as many individuals as processors are evaluated simultaneously. This fact produces an improvement in the time required for evaluating all individuals [6],[7].

Table 1. Some operators and their characteristics.

Operator	Number of operands	Operation
*	2	Par_1*Par_2
AND	2	Par_1 & Par_2

In GP the number and type of instructions utilized is not always the same. Thus, for each instruction, operator name, number of operands involved in the operation and the way operators work with operands to produce the result vary (see Table 1). This information is essential for GP. It builds the trees representing programs composed by the allowed operations for the proposed problem. In a numeric problem, we can find a tree like a tree-like structured as depicted in Fig. 1.

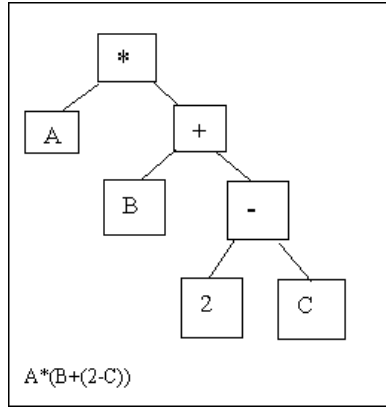


Fig. 1. An Individual represented by means of a tree.

As we have seen above, the number of operators and their definition is given beforehand for a particular problem. A list containing all the nodes that composes the tree can be easily created, as in Table 2, simply by crossing in-order the tree. Starting from this list it is easy to build the original tree by means of the inverse process. In order to do that it is enough to check every component contained in the list. In this way we determine how many descendants each node of the tree must contain. Thus the tree is built without errors, putting nodes and leaves in their correct position.

Table 2. List of nodes contained in the above tree.

Node	*	A	+	B	-	2	C
Number of Descendants	2	0	2	0	2	0	0

In order to exchange individuals between populations we need a function for receiving individuals. This function must take into account the way trees arrive, like an in-order node list. This function must check the number of descendants per node. The sending function will read a tree in-order and will sequentially send every element found (Fig. 2 shows an example).

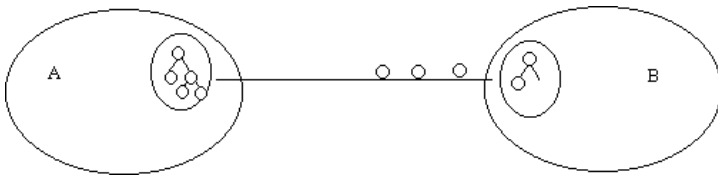


Fig. 2. Population A sends an individual to population B.

Thanks to the PVM's capacity for specifying the process to which a message is directed, individuals can be sent between whatever two populations.

2.1 Population Size

Once the sending and receiving mechanism is established, every population may send to another one as many individuals as desired. However if a population receives individuals without deleting any, then the population will grow outside acceptable limits. To avoid this problem, each time an individual is received, this will replace the worst individual. Thus, the amount of individuals does not change. We must notice that a sending operation really does not move any individual, this operation rather duplicates an individual and sends the copy. This implies that the number of programs in the transmitter population will always be the same.

2.2 Synchronism

We know that PVM supplies buffers for sending and receiving information. The receiving function never waits for any arriving individual. Rather it checks if there is a coming individual, looking in the reception buffer. If there is one, it is received. In the other case the population continues working. With this mechanism a population will never have to wait for individuals. PVM gives us the necessary functions for receiving individuals waiting for them and without any stopping. Thus it is easy to implement the above process.

We have just seen how populations eliminate their worst programs when a new one arrives from another population. It is therefore necessary to know the worst individual in that instant. The best, the worst, and the range of each individual in the population are known when a generation finishes the evaluation of its individuals. Because of this, receptions have to be made when this evaluation has been completed. But it is possible that several individuals are waiting in the buffer to be incorporated into a population. Each time an individual is received the worst one in the population disappears, therefore it is necessary to recalculate the new worst program, which will be deleted the next time an individual will be received. In this way as many individuals as necessary can be received.

3 Implementation

In the code below we show the basic GP algorithm modified with the functions that allow sending and receiving individuals. Several processes following the algorithm can communicate among them sending their best individuals.

```
generation=0

seed population

while not termination condition do
    generation=generation + 1
    calculate fitness
    selection
```

```

crossover (Pcross)
mutation (Pmut)
if send_condition then
    send best individual to a neighboring pop.
end if
while individual to be received
    receive individual
    replace worst individual
    calculate worst individual
end while

end while

```

In the following section we will see how populations really do not decide to whom their individuals are sent. A specific node acting as a mail-office receives all individuals sent from populations, and decides where they must be sent. Thus, the real operation of each population is very similar to the sequential model, although the logic corresponds to that shown in the above pseudo-code.

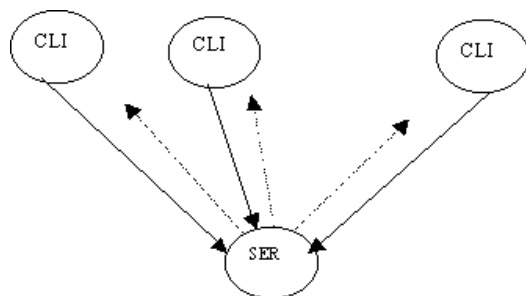


Fig. 3. Client/Server Architecture

3.1 Client/Server Model

We must underline that each population knows that other populations are evolving simultaneously. This knowledge is necessary to decide where to send an individual. Any process, i.e. any population that evolves, must take charge of issuing as many populations as will be necessary for solving the stated problem. We have chosen to use a specific architecture based on the client/server model. The server will take charge of message handling among clients, which are populations. Moreover, it initially issues the populations that work in parallel. Each population has a mailbox where the input-output messages are placed. The server process must work as a post office, taking the output mail of each population and driving them to their destination by putting each mail in the mailbox of the corresponding population (see Fig. 3).

3.2 Topology

We could have allowed populations to select the destination when an individual is being sent. Nevertheless the server carries out this work. The server possesses a table, which indicates for every transmitter the corresponding receiver. This model has the advantage of establishing a new communication topology by only modifying the server's connection table. Populations don't need this information. They only receive individuals from a server and send individuals to this central process. Moreover, since populations produce individuals at a slower rate than the server for sending them again, bottlenecks will never happen.

We can easily use any topology (ring, random, neighbourhood...) by modifying the table mentioned above. We can consider the server as a switching element (Fig. 4)

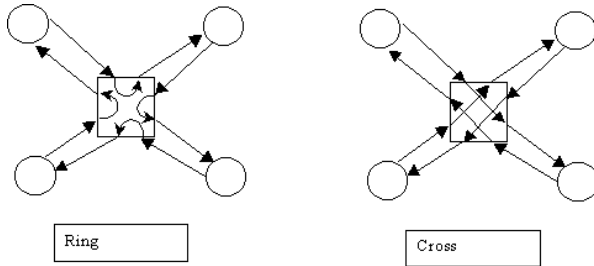


Fig. 4. Communication Models.

3.3 The Generation of Random Numbers

Another aspect taken into account for modifying GPC in order to run several populations in parallel was the generation of random numbers. In any searching process based on GP, it is always necessary to generate an initial random population. To do that, random numbers must be generated.

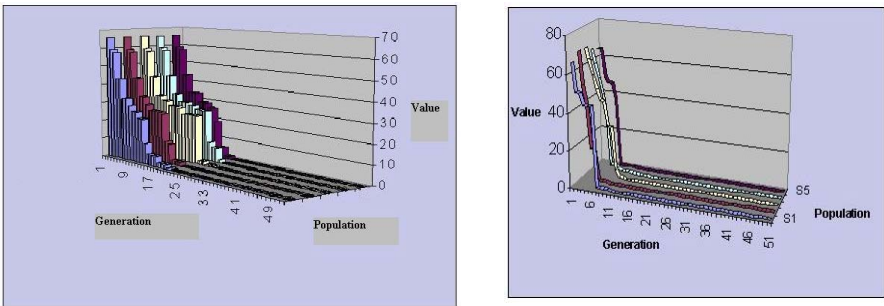


Fig. 5. *Left)* Several populations getting almost the same results. *Right)* Several populations getting different results once random number generation has been corrected.

GPC can produce this kind of numbers, but these numbers are very similar in several populations simultaneously executed in the same processor. Moreover, if due to the communication topology there aren't differences among instructions executed by each population, then the random numbers generated will be identical. Thus, the individual to be crossed or mutated will be the same, because of their dependence on random numbers, and all populations will practically find the same results regardless the number of generation executed, as can be seen in Fig. 5, *Left*.

In order to solve this problem we modified the routine that generates stochastic values. We added to the calculus of random numbers, the number of processes corresponding to each population. In Fig 5 *right*, we can see that each population evolves in a different way generation after generation.

4 Results

In order to compare the performance of GPP versus GP it is necessary to apply both methods to a wide set of problems. We have chosen the "Artificial Ant on the San Mateo Trail" problem. The goal is to find a program for controlling the movement of an artificial ant in order to find all of the food lying along a series of irregular trails on a two-dimensional toroidal grid. The ant's sensory ability is limited to sensing the presence or absence of food in the single square of the grid that the ant is currently facing. The ant's potential actions are limited to turning right, turning left, and moving forward one square. The *San Mateo trail* consists of nine parts, each made of a square 13-by-13 grid containing different discontinuities in the sequence of food. The discontinuities include: single and double gaps, corners where a single piece of food is missing, corners where two pieces of food are missing in the trail's current direction, and corners where two pieces of food are missing to the left or to the right of the current direction of the trail.

According to our model we have a set of clients, which are different ants populations, and a server we call ants' nest, that takes charge of communications.

We have experimented with several communications topologies. In Fig. 6 we show the results using a random topology. In this figure we can see that the convergence process is quicker when there are several populations with communications among them than in the case of only one population. When each population works in a different processor then the required time for applying GPP on n populations is the same as GP needs for one population plus a little increment due to communicative requirements. If we use m processors and n populations, being $m < n$, the time employed to process n populations will continue being small, approximately n/m . When $m=1$ we are developing a parallel process in only one processor. The amount of time consumed will approximately be the same as the amount GP uses, but because of algorithms differences, the convergence process speeds up, and we do not need wait for many generations as GP needs.

5 Conclusions

In this work we have developed a tool, based on PVM, that have proved to be very useful to experiment GPP on distinct problems. We have applied this tool to a particular environment by only compiling the C++ source code with the C++ compiler available, and we immediately can begin the GPP experiments.

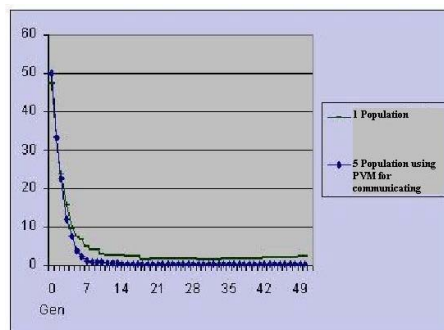


Fig. 6. Five Populations communicating among them with random topology compared with a population evaluated sequentially. (The total number of individuals evaluated is the same in both cases)

References

1. Koza, J.R.: Genetic Programming: on the programming of computers by means of natural selection. Cambridge, Mass: MIT Press. 1994.
2. Darwin, C.: The Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life, Mentor Reprint, 1958.
3. Banzhaf, W., Nordin, P., Keller, R.E., Francone F.D.: Genetic Programming. An Introduction. Appendix C: GP Software. Morgan Kaufmann Publishers. pp393. 1998.
4. Weinbrenner, T.: Genetic Programming Kernel Version 0.5.2 C++ Class Library. <http://thor.emk.e-technik.th-darmstadt.de/~thomasw/gpkernel1.html>
5. Arnone, S., Dell'Orto, M., Tettamanzi, A., Tomassini, M.: Highly Parallel Evolutionary Algorithms for Global Optimization, Symbolic Inference and Non-Linear Regression. Proceedings of the 6th Joint EPS-APS International Conference on Physics Computing. 1994.
6. Oussaidène, M., Chopard, B., Pictet, O. V., Tomassini, M.: Parallel Genetic Programming and its application to trading model induction. Parallel Computing, 23, pp. 1183-1198, 1997.
7. Andre, D., Koza, J.R.: Parallel Genetic Programming: A Scalable Implementation Using The Transputer Network Architecture. In P. Angeline and K. Kinnear editors, Advances in Genetic Programming 2. The MIT Press, 1996.

Net-Console: A Web-Based Development Environment for MPI Programs

Andreas Papagapiou, Paraskevas Evripidou, and George Samaras

Department of Computer Science
University of Cyprus
P.O. Box 537, 1678 Nicosia, Cyprus
{anpap,skevos,cssamara}@ucy.ac.cy

Abstract. In this paper we describe Net-console, a Web-based environment for the development of Message Passing Interface (MPI) programs. Using Net-console the user is able to write, execute, debug and evaluate the performance of such programs across the Internet, using a usual Java-enabled browser, without having to log on to a supercomputer in the traditional sense. The tools included in Net-console and their functionality, the languages used and the overall structure of the project are presented in this paper.

1 Introduction

This paper presents Net-console, a suite of tools for the development of MPI programs across the Internet. It is used as the user interface for high-performance computing (HPC) sites. The user is able to upload, create, edit, execute, debug and monitor MPI programs, using a standard Java-enabled browser from anywhere, anytime. Net-console is developed mainly in Java, with CGI scripts used to perform some specific functions.

Over the last few years we have witnessed the rapid growth of the Internet and the wide use of the services it provides. Undoubtly the most widely used service is Web page browsing. Web browsing tools exist for all platforms, making the Web accessible from every kind of computer. Lately with the introduction of Java [5] - a complete object-oriented programming language - complicated applications can be created and used through the Web.

The use of parallel processing is essential in cases high computing power is required. Nevertheless, development of MPI [8] programs can be proved to be very tedious, especially when X-based tools (for editing, debugging, performance monitoring etc.) cannot be used. However, it is possible to utilize the power of Java and Web browsing tools to create a portable and user friendly interface for performing the development of such programs.

In this paper we propose the use of Java, CGI and Web browsing tools in the development of a user friendly environment for creating MPI programs from anywhere in the Internet. This is an integrated environment that can be used as a front-end to HPC sites. No direct shell access nor a shell account on the HPC are needed in order to use it. The environment consists of an editor, an execution console, a debugger, monitoring tools and an Account and File Manager.

2 Architecture and Structure

2.1 Web Server and User Accounts

In order to use Net-console the user needs access to the Internet and a Java-enabled Web browser. Net-console must be hosted on a Web server. Applets, HTML pages, CGI [3] scripts and user files are stored on this server. Programs needed for Net-console to function, such as gdb, compilers and libraries are also stored there. For the prototype we have developed we used a Unix machine as a Web server and the HPC facility is a Network of Workstations that belong to the same LAN as the server. Each user of the system is entitled to a local directory in which his files are uploaded, saved and manipulated. When the user first accesses one of Net-console's pages, he is asked for a user ID and password. These are logged and used in every following script to determine the local directory and files to be accessed by the corresponding user. With the use of the Account and File Manager, the user can upload, compile, list or delete files and change his password. This of course, is the password of the user used by the Web server to authenticate the remote user.

2.2 Security

For access control we use the Web server's authorization mechanism. We consider it secure to have passwords transferred through the Internet since new browsers support encryption of such information before transmitting them. Directories containing applets, CGI scripts and HTML pages are protected with .htaccess files. Hence even if the browser is pointed directly to one of the protected directories, the user will still be asked for a user name and password. The only pages visible without a password are the ones describing how to gain access to the system and the main Net-console home page. Precautions are taken so that the user is not able to access files on the server others than the ones in his account. Also, as an additional security measure the source code is checked so that it does not contain commands that have direct access to the operating system (such as "system" and "getenv" in C). In addition uploading of executables is not allowed. These checks are performed automatically by the upload and compile CGI scripts. Furthermore, user's actions are kept in a special log, in addition to the one kept by the Web server. For better security we use a dedicated account (Net-console account) for hosting the environment. The Web server used is being run from the Net-console user, so there is no need to give permissions to "all users" in order to enable access to the pages or applets. (This is necessary when running the Web server as "nobody" or "webuser") Also, all the CGI scripts called by it have read, write and execute permissions to the account. That way there is no need to use "setuid" to gain these privileges.

3 Net-Console Components

Net-console consists of an editor (Net-med), an execution console (Net-xec), a debugger (Net-dbx), a monitoring tool (Net-mon) and an Account and File Man-

ager. The environment consists of Web pages, some of which provide an interface to CGI scripts and/or Java applets. The user interface used in applets is very much like the ones we are used to seeing in application programs, contributing to ease of use and understanding.

3.1 Net-med

Net-med (**Net-MPI editor**) is a Java powered, MPI-aware text editor. It enables the user to easily create or edit files in his account. That way the user doesn't have to upload a new version of the file every time a correction is needed. All the processing is done on the client, since the applet runs locally. After the applet is downloaded and run, it only needs to communicate with the server for open, save and compile operations. Apart from the standard editor capabilities to ease editing, it provides specific MPI related functions. When new file is created the standard MPI skeleton (i.e. standard include commands, `MPIInit`, `MPIFinalize`) is automatically inserted. It provides menus for all the MPI command types and their variations. The user simply selects the command to insert and a new form appears. The user can complete the form either by typing the parameters in the text fields or by copying the selected text from the editing area into them using the mouse. There is also the option to insert a profiling command, or commands supported by the `MPLFT` [10], a library written in C that supports fault detection and recovery of MPI processes. The MPI commands can be found grouped, with help available for each one of them. Detailed help on how to use the editor is also available. In addition, Net-med provides Send-Receive matching. This can be done directly from the insert send form, or by highlighting the `MPISend` in the code and using the "match receive" function provided in the menu. In both cases a form to insert a Receive will appear, with the corresponding fields completed.

The user can also compile the code being edit and watch the compiler's output in a separate frame. Graphical shortcut bars for the most frequently used operations are available and can be visible or hidden, according to the user's preference. Net-med also offers a print function. This opens the source file in a Web browser window and enables the user to print it through the browser's Print option. In addition to this, a simple lexical analyzer can be used to format the source file as an HTML page and display C commands, MPI commands, comments etc in different color, making the source easier to understand.

Figure 1 depicts a screen shot of Net-med's prototype that we have developed. On the top of the page exists a navigation frame that allows us to switch between Net-console's tools. On the top left of the screen shot we can see the main editing window with its menu. All functions are called through this menu. The internal window in the middle, titled "MPI commands", has most MPI commands grouped by type. The types of commands appear in the list on the left and when one of them is clicked, all the commands of that type appear in the list on the right. The user can then display the insert-command form (top right) by clicking on the *insert* button or by double-clicking the command. The syntax of the command is displayed on the top of the form. The most common MPI

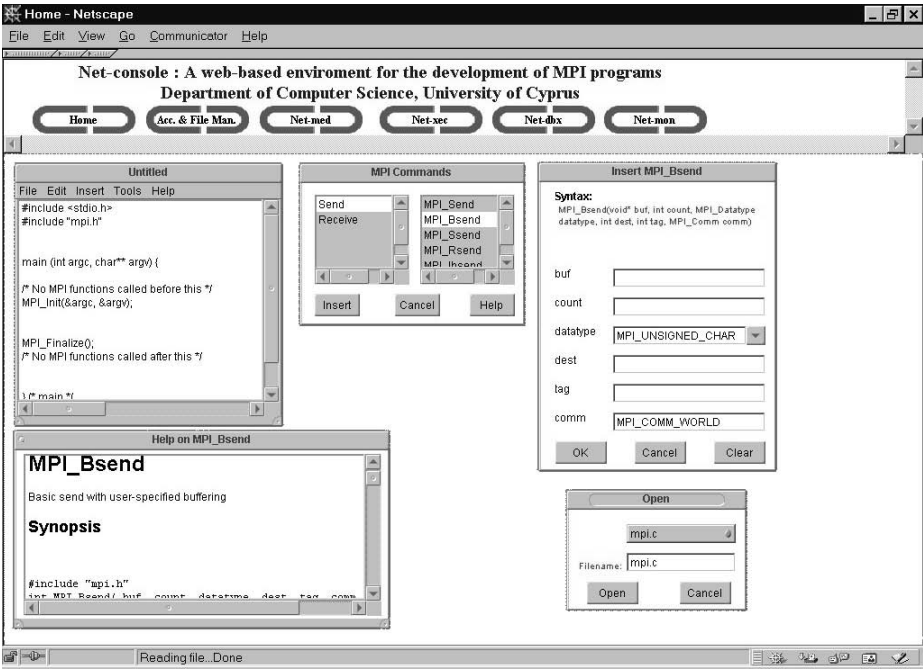


Fig. 1. Screen shot of Net-med.

Datatypes are available in a combo box in the datatype field of the insert form. The most commonly used communicator (MPI.COMM_WORLD) is displayed as default at the communicator field. Net-med provides a function for creating new communicators. The names of the communicators created with this function are available in a combo box in the *comm* field of the form. The user can select the communicator name from the combo box, type it or copy it in the text field by highlighting it in the editing area and double clicking in the field. Clicking the *Help* button will display the (closeable) help window for the specified command (bottom left), while clicking the *cancel* button will close the “MPI commands” window, leaving the insert and help windows open.

The open file dialog is visible in the bottom right. The user can type the file-name to be opened or select it from the list of available files in the popup menu provided. The files are listed with the aid of a CGI script and inserted into the popup menu.

A feature of Net-med we don’t usually find in Java applets, is its ability to save and retrieve files from the server on which it resides. This - due to Java security restrictions - cannot be done by applets. There is, however a simple way to overcome this problem [2]. Java applets can accept input from or output to CGI scripts. In order to open a file Net-med, after getting the filename from the user, opens a URL input stream. This URL is the location of the script that performs the open operation, called with the selected file as a parameter and

the script opens the selected file. In the prototype we have developed this is done by using the Unix “cat” command. That way, we have the file contents passed to the input stream opened by the applet. The applet reads from the input stream, line by line and appends each line read in the editing area. Saving files is done in a very similar way. This time the applet uses the URL of the CGI script (with the filename to be saved as a parameter) as an output stream and the contents of the editing area are passed to it. The CGI script writes the data received in the filename given as a parameter. The user can only read or write files in his account. To ensure this, the account’s path is kept into a system variable for every user. This variable gets its value after authorization for the user is completed, using the *remote_user* variable created. It is used in the scripts, so that the files are stored in the correct account. Listing of available files mentioned earlier is achieved in a similar way (by using the URL of a CGI script as input stream).

3.2 Net-xec

After the user has written and compiled a program, he can execute it through Net-xec. The first step is to choose the executable from the list of already compiled programs. The user can set the number of processing nodes the program should run on. After confirming these settings Net-xec connects through telnet to node 0 and initiates the execution of the program.

As soon as the program starts executing an interaction terminal will appear. This is an area where the user can see the output of the program and give data from the keyboard if needed. Once the program is running the user has the ability to watch the percentage of processor use for that program, on some or all the processing nodes. This can be achieved by using the Processor Utilization Monitor, integrated in Net-xec. The user has the ability to choose the nodes to monitor and which of these nodes to provide information at any moment. The Processor Utilization Monitor provides control on separate processing nodes as well as group control. The processor utilization is displayed on a progress bar for every monitored process. Peak and Current values are visible to the user who is able to change the monitoring update interval. The processor utilization is obtained from each processing node by using Java telnet classes to connect to it and Unix commands to get the percentage of processor use for a specific process. Due to Java security restrictions, all connections must be done through the machine on which the Web server resides.

3.3 Net-dbx

Parallel programs differ from serial ones mainly in the way memory is used and in program flow. MPI programs are difficult to debug, because of the distribution of processing. To find a bug the user has to monitor execution on every node and also watch all the pending messages. Net-dbx [7] is a tool for distributed interactive debugging across the Internet. It offers a graphical user interface to gdb (GNU debugger), a source code level runtime compiler, which provides facilities

such as changing the variable values during execution and setting breakpoints on certain conditions. The gdb can be attached to a running Unix process and debug it using the information provided by the core and map files generated during compilation with the `-g` option. To achieve distributed debugging Net-dbx attaches to every process at the OS level and then interacts with the debugger using a user friendly interface. The user defines global and individual operations to be applied to a process or a group of processes. These are translated by the interaction tool to interaction with the local debuggers individually. Net-dbx's interface includes features such as run-time source code level debugging, variable watch, expression evaluation, breakpoint setting and step-by-step execution on a single process or over a group of processes.

Net-dbx and Net-xec use a special initialization scheme so that they get the information they need about the individual processes. This is done by embedding an initialization routine in the code, at compile time, transparently to the user. This routine, at the beginning of the execution, provides data about the process id and the processor name for every process. These data is passed to the applets and processed to extract the information needed. Depending on the arguments with which it is called it will either wait until all the processes are attached to the local debuggers (Net-dbx) or continue execution (Net-xec).

3.4 Net-mon

Need of performance visualization is essential for evaluating the performance of large MPI programs in order to improve them [9]. Net-mon provides information about the interprocess communication time (e.g., time needed for a processor to send or receive a message) and virtual memory statistics, such as virtual memory use and page faults. For such information to be generated, timing commands must be included in the code. In order to ease program development, this can be done transparently to the user. Using a lexical analyzer, specific MPI commands are replaced with corresponding commands that perform the same function, but also keep track of the time elapsed for the completion of the actual command. All of these profiling commands are included in a separate library. With the aid of a CGI script a copy of the program is created with some or all of the MPI commands replaced by our profiling commands. The copy of the program is then compiled and linked with the profiling library and executed through Net-xec. Every node stores its traces in a file. Since the program is running simultaneously on more than one machines, we cannot use a common file for storing the output. Instead we use a local file, accessible by each processor individually. A unique filename (containing the process id) is used to avoid conflicts in case multiple users are running a program simultaneously or even if more than one process is running on some machines. These files are created by the profiling Init command and moved into the user's directory by the Finalize command when the program terminates. The user is then able -with the use of another Java applet- to view the trace files separately, or combine them into a global trace file. A search feature is available to enable the selection of traces with certain characteristics, like send and receive time, source or destination. Tracefiles

are created in HTML format, using various colors, to make understanding them easier. Graphical representation of the trace files is also available. The different communication commands are presented with different colors, and messages are presented with arrows from source to destination.

3.5 Account and File Management

Account and file management functions are performed by simple CGI scripts and enable the user to upload, compile, list or delete files and change his password. The upload utility stores the uploaded files in the user's directory. The compile utility can be used directly from the utilities page or through Net-med's "compile" function. In both cases source files from the user's directory are compiled and linked with the Net-dbx library [7]. The compiler output is displayed in both cases. If the compilation succeeds, the executable is stored in the user's directory, with an "x" appended to the source file's basename (e.g. prog1.c compiles to prog1x).

4 Implementation Issues and Portability

Java's ability to run through a Web browser makes it the most suitable language for our purpose. In addition, its multi-threading capabilities provide solutions to many problems encountered during the design and implementation. For the Java applets, Netcape's IFC classes [4] were used for building the user interface. The most important reason for using IFC is their platform independent look-and feel and the TextView component they provide. TextView supports multi-formatted text, automatically wraps lines and has the ability to import and format HTML. It also features a text filter interface that can process every character before being displayed. These capabilities make it ideal for text viewing and editing area as well as for displaying HTML files (such as help and trace files). A set of telnet classes [6] was used for connecting to the nodes and running the programs or getting information about them.

CGI scripts were also employed to perform functions that couldn't be performed by the applets alone. Due to Java's security restrictions, applets cannot read from nor write data to the local (Web-server's) disk. Simple CGIs were used to implement the "open" and "save" functions of Net-med. CGIs were also used for listing the user's directory contents through the applets, as well as for the Account and File Manager (upload, dir, delete, compile, change password). The lexical analyzer generator "lex" was used for building the lexical analyzers that perform the function of inserting profiling commands into the source code.

Java's platform independency enables us to install Net-console on virtually every platform. For the prototype we have developed we used C for the upload script and Unix shell's scripting abilities for the rest of the scripts. As long as Net-console is hosted on a Unix machine, these will work with minor or no changes. For every Java-based tool in Net-console there exists a class file containing information such as CGI script's location, help URL, user names and

passwords. These are the only files that will need editing and re-compilation when installing the environment. Web server directories and user accounts can be set up in any way considered convenient.

5 Concluding Remarks and Future Work

In this paper we have presented Net-console, a suite of tools that utilize Web browsing tools and Java capabilities for MPI program development from anywhere in the Internet. The working prototype we have developed proves that teleworking can be done through the Web. This environment provides the capability to develop programs through the Web using a graphical, user-friendly interface. It can be used from anywhere in the Internet, even with a low bandwidth (33Kbps) dial-up connection. However, the most important feature is the portability of the whole environment, resulting from the use of Java for the implementation. The rapidly increasing use of Java, Web browsing tools and the Internet, as well as dedicated Java hardware announced (Java chips) give us reasons to believe that such environments will be widely used in the near future.

We are currently working on improving our prototype by adding more functions to the tools. We are planning to improve some functions of the tools by using mobile agents to handle asynchronous events. Our next goal is to use the capabilities provided by Globus [1] for integrating geographically distributed computational and information resources for parallel processing, with Net-console as the user interface.

References

1. "The Globus project.". <http://www.globus.org>.
2. Scott Clark. *"Java Jive: File I/O with Java: It can be done"*.
http://www.webdeveloper.com/java/java_jj_read_write.html.
3. Rafe Colburn. *"Teach Yourself CGI programming in a week"*. Sams.net Publishing, 1998.
4. David Flanagan Dean Petrich. *"Netscape IFC in a Nutshell"*. O'Reilly, 1997.
5. Bruce Eckel. *"Thinking in Java"*. <http://www.Eckel0bjects.com>.
6. Matthias Jugel, Marcus Meiner. *"The Java Telnet Applet: Documentation"*.
<http://physlab.catlin.edu/oldmiscstuff/Telnet/Documentation/index.html>.
7. Neophytos Neophytou, Paraskevas Evripidou. "Net-dbx: A Java powered tool for interactive Debugging of MPI programs across the Internet.". In *Proceedings of Euro-Par 98 conference, Southampton, UK*, September 1998. <http://www.cs.ucy.ac.cy/~net-dbx>.
8. Peter Pacheco. *"Parallel Programming with MPI"*. Morgan Kaufmann, 1997.
9. Titos Saridakis. "Array Tracer: A Parallel Performance Analysis Tool". Master's thesis, University of Crete, September 1995.
10. Soulla Louca, Neophytos Neophytou, Adrianos Lachanas, Paraskevas Evripidou. "MPI-FT: A portable fault tolerance scheme for MPI.". In *Proceedings of PDPTA'98 International conference, Las Vegas, Nevada* 1998.

VisualMPI – A Knowledge-Based System for Writing Efficient MPI Applications

Dariusz Ferenc, Jarosław Nabrzyski, Maciej Stroiński, and Piotr Wierzejewski

Poznań Supercomputing and Networking Center
ul. Noskowskiego 10, 61-704 Poznań, Poland
naber@man.poznan.pl

Abstract. Message Passing Interface (MPI), a communication library for both parallel computers and workstation networks, has been developed as a proposed standard for message passing and other related operations. It is aimed to provide the parallel programming community with the portability, scalability and efficiency needed to develop applications and parallel libraries for efficient use of current and future high performance systems. New standards, such as MPI, require new tools which will help to use them. In this paper we present Java based case tool for writing MPI programs. In general a case tool is a computer-based product aimed at supporting one or more software engineering activities within software development. As we show in the paper our system offers automatic MPI code generation features for both C and Fortran assuring high efficiency of the application. Support of expert system is a significant feature of the system.

1 Introduction

High performance computing potentially offers many helpful environments, methods and tools to both, computer engineers and computational scientists. However the use of these modern technologies is still very difficult. Moreover the rapid advances in computing technologies are not making the thing easier - it is very difficult to be up to date with these technologies even if these technologies are very helpful. Message Passing Interface (MPI) is one of such modern technologies. MPI [1] is de facto standard for writing parallel application programs and libraries. By now several tools have been developed for users to help them visualize MPI programs, mostly in post execution mode using some trace files [2], [3], [4]. Other tools allow user to debug his/her programs [5]. The existent tools are designed specially for advanced users of MPI and they require a good knowledge of message passing and related techniques. The design phase of the software development is very important for the efficiency of the program.

There are several tools for making this phase easier for programmer, but not to many of them are designed for MPI. In general these tools are divided onto two groups: textual and graphical ones. Linda is the example of textual representation of parallelism. Such tools as HENCE [7], PARSE [8], GRADE [9], TRAPPER [10] are

the examples of the tools that use graphical representations of parallelism. Most of them use various kinds of graphs. However they differ from each other. HENCE graphs for example describe the behavioral aspects of the computation using control flow oriented model, where the nodes represent computations and the arcs represent dependencies between the computations.

GRADE and TRAPPER use graphs to describe structural aspects of the parallel applications. In their graphs the nodes represent processes and the arcs represent the communication channels between them. GRADE is designed for PVM. Moreover the GRAPNEL language for parallel applications makes it more difficult to get used with the system. TRAPPER process graphs allow only point-to-point communication between processes with no distinguishing between synchronous and asynchronous communication channels.

More expressive graphs can be found in PARSE, which allows 1-to-1, 1-to-n (multicast) and 1-to-all (broadcast) communication structures on synchronous or asynchronous communication channels. PARSE uses Petri nets to describe, model and verify the communication and synchronization of processes. Systems designed using PARSE are represented using a graphical notation known as process graphs, which enable process structures and their precise interactions to be hierarchically constructed. However, the current PARSE graphical notation cannot adequately handle the design of concurrent systems which incorporate dynamic features.

In this paper we present Java-based, environment, namely VisualMPI, designed for programmers who want to write parallel distributed applications without possessing a good knowledge about message passing. The important aspect of the tool is its ability to teach users how to use different MPI functions. Its knowledge based and automatic performance design allows to generate efficient MPI codes for both C and Fortran. These intelligent features make this tool very interesting also for advanced MPI programmers.

2 Knowledge-Based Concept of VisualMPI

VisualMPI is a graphical environment, written in Java, that helps users to write parallel applications with MPI. It supports such stages of the application design as code development, application testing, monitoring and optimization. VisualMPI is a knowledge based system. It means that all the stages are supported by some knowledge experts. The knowledge in the system is represented in a form of objects and rules. These objects and rules describe different kinds of models, such as target architecture models, performance models, application models, bottlenecks of the software and hardware environment.

Our aim was to build a system that would integrate the concepts from expert systems (ES) and decision support systems (DSS) to produce a successful development aiding system that explicitly considers the role of performance of the application. So, by combining both ES and DSS into one system one can describe the application in a symbolic manner (easy to understand for the programmer) and to take into account different models of applications and architectures to produce an efficient MPI based application code.

2.1 System Architecture

In our system the application development process is thought as an interactive co-operation of the programmer with the tool through the selection and sequencing of processes, mapping them to processors, running the application with control of the system (tracing), finding the application bottlenecks and resolving them to achieve the problem objective. The objective here is always the same – obtaining efficient application being able to run on the target platforms.

VisualMPI is composed of several modules (Fig 1.): (1) expert system which runs as a kernel of the whole system, (2) graphical code design tool, (3) code generation module, (4) application tuning tool and finally (5) visualization tool. All these modules are responsible for different design phases of the application. The program design cycle is the following:

- Hardware and software design in a graphical way,
- Knowledge-based code generation,
- First execution and performance optimization,
- Monitoring,
- Visualization.

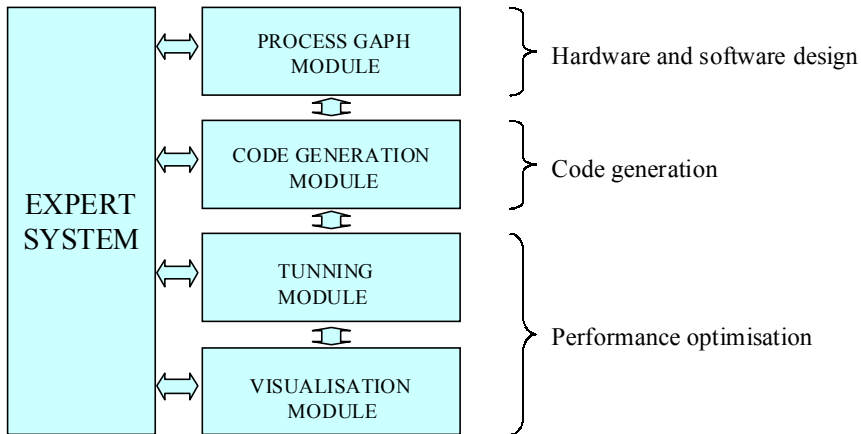


Fig. 1. General system architecture and program design stage

As one can see the expert system plays the main role here. Below we present the main features of different modules and their role in the MPI application development.

2.1.1 Expert System

The expert system which controls the behavior of all the modules is based on the blackboard architecture. The idea of such an architecture is quite simple. The entire system consists of a set of independent modules, called knowledge sources (KS's), that contain the domain-specific knowledge (knowledge base and inference engine) of the system, and a blackboard - a shared data structure to which all the KS's have access (Fig. 2). Such an architecture allows to place various experts in a distributed system and there make them waiting for being activated.

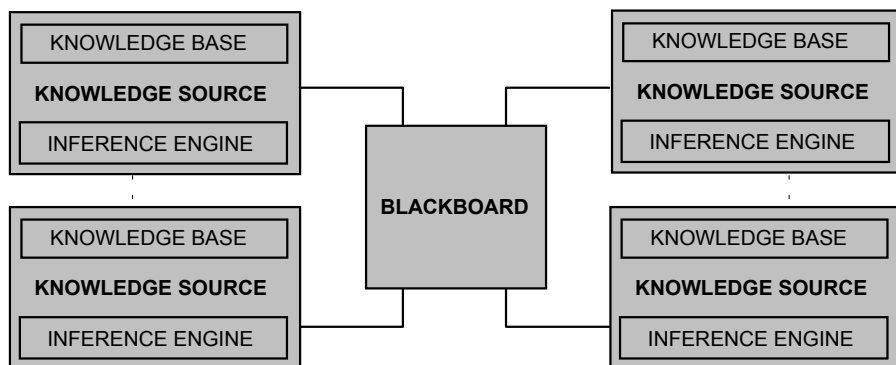


Fig. 2. Blackboard architecture of the expert system

When a KS is activated it examines the current contents of the blackboard and applies its knowledge either to create a new hypothesis and write it on the blackboard, or to modify an existing one. Associated with each KS is a set of triggers that specify conditions under which the KS should be activated. These conditions are determined by particular experts. As mentioned before there are several types of experts in the system. Each expert is responsible for different aspects of distributed programming with MPI (programming tricks). There are also experts that possess the knowledge about current state of different machines in a metacomputer environment. Such knowledge can be divided into the static and dynamic ones. The static knowledge is the one which has been stored in the past and is not changing. The dynamic one represents the changeable behavior of the metacomputer.

Each KS contains:

- activation conditions,
- knowledge base,
- inference engine.

KS can use one of three types of inference methods: forward, backward or mixed reasoning. Because there are different experts co-working on one problem (represented by the KS) every KS has its own independent inference engine. A blackboard, i.e. a data structure that is available to all experts, serves for communication and stores the active knowledge of the state of the application development process. The blackboard carries problem observations (symptoms, i.e. objects with their values), local abstractions (data abstractions with their values), hypotheses and intermediate and final solutions. All MPI features are known from the beginning and are incorporated into the system as a knowledge bases and the initial contents of the blackboard. When a KS is activated, a particular expert examines the current contents of the blackboard and applies his knowledge either to create a new hypothesis and write it on the blackboard, or to modify an existing one. Observations and data abstractions can be read by all the problem solving experts. We have two typical interactions:

- certain experts write suggested hypotheses on the blackboard which are tested by other experts if there is no contradictory knowledge about the same situations,
- the experts write different intermediate and final results on the blackboard and the decision between them is taken on a global level.

Of course different parts of the knowledge base are used for co-operating with various modules of the system.

To represent a domain knowledge one needs to choose the type of the knowledge representation and a special formalism to introduce the knowledge into the system. We have decided to use the well known representation based on the frames and rules. Frames are serving for describing different objects and rules operate on these frames causing some actions.

There are some specialized languages for knowledge representations available, just to mention Schema Representation Language (SRL) or Nested Interactive Array Language (NIAL). However these languages have some disadvantages. In the first place they are platform dependent and in the second they are not object oriented. Our system is implemented in the heterogeneous environment of metacomputer at Poznań Supercomputing and Networking Center. This metacomputer consists of SGI (PowerChallenge, ONYX2, workstations), CRAY (T3E and J916), IBM RS6000/SP2. Thus it was necessary to define a specialized meta-language which would be platform independent. This meta-language is used for knowledge representation and defining the knowledge sources, the blackboard and other mechanisms of the system.

The hierarchy of abstractions which is made possible by frame-based representation provides a good model of the problem while decomposing the domain knowledge into reasonable, manageable pieces. There are the following object types defined in the system:

- **Observations and Data Abstractions.** Observations correspond to the questions of heuristic classification and serve for primary data collection. Data abstractions are derived from the observations with simple rules.
- **Observation Classes.** They group together observations and data abstractions and serve for the simplification of the dialog control or for assigning solutions to groups of data abstractions. They include the question classes of heuristic classification, the materials of the functional models and the set of the parameters of a state in fault models and combine their attributes,
- **Intermediate and Final Solutions.** They represent the current state of the programming process and include the combination of the attributes of the solutions and states of all the KS's. In particular, they can be valued differently by each KS,
- **Rules and Constraints.** They express the knowledge of those object-object relationships which cannot be expressed by direct object references. Rules and constraints are connected to the objects concerned by a system of pointers. Thus, inference process can be implemented very efficiently by message passing between the objects. This requires that all the objects appearing in the rules must be known.

An object has only parts or refinements as its successors. A variable number of parts is allowed, but may be used only if the number of parts is of no great importance, so that a case distinction with specializing objects is not required. The number is represented as the parameter "number" of the part. This has the consequence that all parts must be refined in the same way, since they are represented as a single object.

The refinements of objects have the same parameters as their predecessors, but their parts usually have a different parameter structure. The specializing hierarchy therefore corresponds to the usual frame structure with the inheritance, attached procedures and default values. However, the parameters are represented as independent objects rather than attributes, the objects having pointers to their parameters, so that we require only a triple representation (object-attribute-value + parameter-attribute-value) instead of the quadruple representation (frame-attribute-facet-value). A parameter belongs potentially to several objects of a specializing hierarchy. For each refinement of an object the current value range of a parameter may be further restricted.

2.1.2 Process Graph and Code Generation Modules

Process graph module (Fig. 3) allows a user to "draw" his application with a use of different graphical symbols. The concept here is similar to TRAPPER. The only difference is the assistance of the expert system. Expert system gives a user a guidance when it comes to choosing different process topologies for a given problem and mapping it onto a hardware.

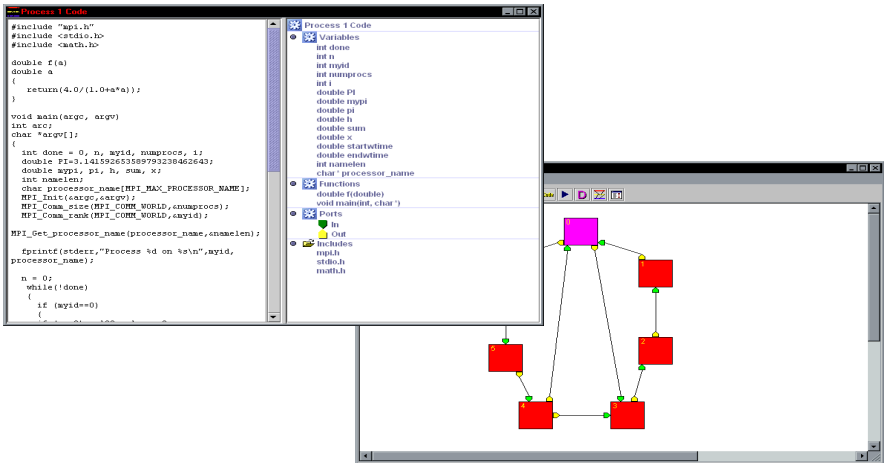


Fig. 3. Process graph module and the program code

Expert system contains the models of possible hardware architectures and is able to proceed with efficient mapping of a software to hardware. The dynamic knowledge of the expert system keeps it informed in all the possible changes of hardware environment, including communication parameters.

Visual MPI generates a program or program template, depending on the amount of the information provided by the programmer. In general a programmer can “draw” all the application, but of course for day to day programmers it is not convenient. Thus, mostly the user receives a template with pointed fragment where a code can be inserted. Expert system always displays to a user only the data (variables, constants, procedures) that are of his interest (Fig. 3, upper right window). User can browse all these data and the expert system will show all the dependencies between them. This a feature that most of the users like. In spite of the template a user gets all the explanation on a WHAT-WHERE-WHY basis in a similar way as in the case of Paradyn system [10]. It is very helpful for both advanced and beginner users. Even advanced users can learn a lot about the performance tips that can be undertaken.

When the application is ready it can be run on the live system. Here we come to next stage – performance tuning.

2.1.3 Performance Optimization

Performance optimization stage offers to programmer two options – auto optimization or manual.

Auto optimization is based on the “black box” features, that are hidden from the programmer. Of course after the optimization has been made, a user can view all the changes made by the expert system with deep explanation of them. A user has also a visualization tool (VAMPIRE-like) that shows all the inter-process communication parameters. (Fig 4.)

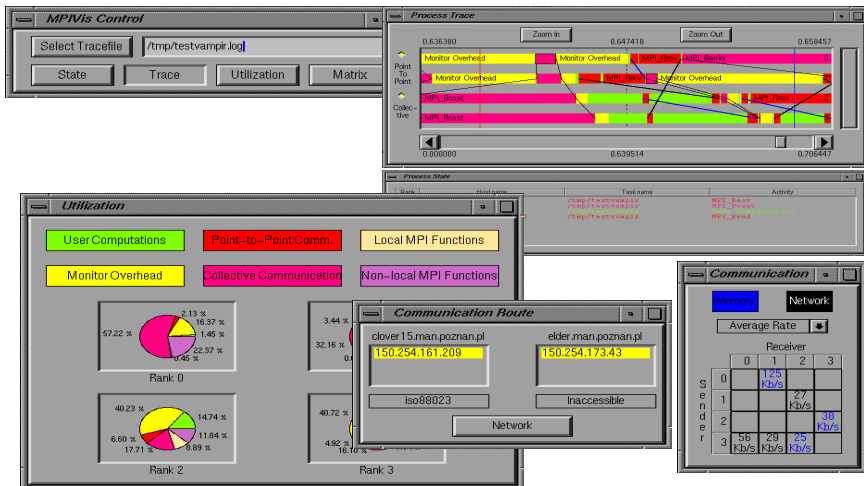


Fig. 4. MPIVIS – MPI visualization tool

Although similar to VAMPIR a tool has some additional features. It allows a user to find all the communication and computation bottlenecks automatically. So, even beginner users will easily find the bottlenecks. Moreover they will get detailed

procedures how to remove these bottlenecks. This feature is available only in a manual mode.

Each optimization process changes or just adds facts to the knowledge bases of different KSs. This mechanism, called automatic or machine learning, improves a system from usage to usage. The process of knowledge acquisition and learning is very important. The better state of the knowledge the better performance of the application developed with the system.

3. Final Remarks

We have presented a tool called Visual MPI. We have focused on a system architecture and main feature that make it different from other similar tools for application development. Knowledge-based user guidance is the main such feature. This feature gives the best results with programming for heterogeneous systems, like for example metacomputing environments, where different options of software to hardware mapping are available.

The early experiences with the tool, which is still under development and improvement, showed the importance of the correct, not contradictory knowledge about the MPI programming model and different bottlenecks of underlying target architecture.

Thus, future work will be focused on methodology of knowledge acquisition for metacomputing environments and finding new methods for bottleneck detection.

References

- [1] Message Passing Interface Forum. MPI: A Message Passing Interface Standard. International Journal of Supercomputer Applications 8, 1994. Special issue on MPI
- [2] <http://www.pallas.de/pages/vampir.htm>,
- [3] <http://www.mpi.nd.edu/lam/>,
- [4] J. Nabrzyski, M. Stroński, MPiVIS – Visualization of MPI programs, Report 12/1996, Poznań Supercomputing and Networking Center Technical Reports,
- [5] <http://www.etnus.com/tw/tvover.htm>
- [6] A. Beguelin, J. J. Dongarra, G. A. Geist, R. Manchek, and V. S. Sunderam. Graphical development tools for network-based concurrent supercomputing. In Proceedings of Supercomputing 91, pages 435-444, Albuquerque, 1991.
- [7] I. Jelly and I. Gorton "The PARSE Project" in Proc IFIP International Workshop on Software Engineering for Parallel and Distributed Systems, March 1996, Berlin, Germany, Chapman and Hall (1996)
- [8] [9] GRADE: A Graphical Programming Environment for Multicomputers, P.Kacsuk, G. Dózsa, T. Fadgyas, R. Lovas, Computers and Artificial Intelligence. 17 (5) :417-427. (1998)
- [9] L. Schäfers, C. Scheidler et al. Software Engineering for Parallel Systems: The TRAPPER Approach, In: Proceedings of the 28th Hawaiian International Conference on System Sciences, January 1995, Hawaii, USA
- [10] B.P. Miller, M.D. Callaghan, J.M. Cargille et al. The Paradyn Parallel Performance Measurement Tool, IEEE Computer, vol 28, No. 11, Nov, 1995, pp. 37-46.

Solving Generalized Boundary Value Problems with Distributed Computing and Recursive Programming

Imre Szeberényi¹ and Gábor Domokos²

¹ Department of Control Engineering and Information Technology,
szebi@iit.bme.hu

² Department of Strength of Materials,
domokos@iit.bme.hu

Technical University of Budapest, H-1521 Budapest, Hungary

Abstract. A wide field of technical problems can be described by boundary value problems (BVPs) associated with ordinary differential equations (ODEs). We show a numerical method and its distributed parallel implementation for solving BVPs. We generalize our method presented earlier in [5] to a recursive scheme, capable of solving more complex technical problems, such as global search for stability boundaries. In this paper we also present the efficient, PVM based implementation of our recursive algorithm.

1 Introduction

A wide class of technical problems can be described by boundary value problems (BVPs) associated with ordinary differential equations (ODEs). The applications range from rod theory to control problems in robotics and bio-mechanical models. The common feature of these problems is that all relevant quantities depend on a single scalar variable (this variable might be interpreted as time or spatial arc-length, depending on the application), and the values of the mentioned quantities are prescribed for several values of the independent variable.

The solution of parameter-dependent BVPs is a hard task, especially if we are interested in *all* solutions and do not have any a priori information on them. Such a task is called a description of the global equilibrium path. Most existing methods are restricted to compute some solutions at a fixed value of the parameter and following these solutions as the parameter varies. These methods are commonly called path-continuation techniques and they rely on iterative subroutines which may or may not converge.

In [5] we proposed a radically different approach. This paper deals with the generalization of the results presented in [5]. We start by reviewing the original problem in sections 2 and 3, and then proceed to describe the generalized, recursive algorithm in sections 4 and 5.

The method suggested in [5] is capable of computing *all* solutions without applying iterative steps. To reach this goal, basically two things are needed:

1. a new approach providing a finite-dimensional embedding for the global equilibrium path
2. an efficient parallel code, enabling us to gain from the huge computing power of parallel machines.

Although the choice of the coordinates in the phase space of the BVP is very important, in this paper we emphasize on the parallel version of the PL algorithm called Parallel Simplex Algorithm (PSA). In section 2 we briefly describe the Simplex Algorithm while section 3 describes the parallel implementation. The section 4 outlines an enhanced version of PSA for applying more complex problems eg. finding local minima and maxima of the global equilibrium path; this version relies on a recursive algorithm. Our parallel application was successfully used in different technical problems in structural mechanics, fluid dynamics, biomechanics. In section 5 we briefly display some of these results from performance aspects. We tested the code both in heterogeneous environment (distributed network) and on a 140-node IBM SP2 at the Supercomputing Facility of Cornell University.

2 Overview of the Simplex Algorithm

As we mentioned in the previous section the problems we are interested in can be reduced to the solution of the following equation system:

$$f_i(x_1, x_2, \dots, x_n) = 0 \quad (i = 1, 2, \dots, n-1) \quad (1)$$

where f_i denote (nonlinear) scalar functions with n real variables. In an n dimensional space the $n-1$ component-functions determine $n-1$ hyper-surfaces. The intersection of these hyper-surfaces determines (typically) one-dimensional manifolds (lines) in the n dimensional space. To find these lines, i.e. to solve this equation system Domokos and Gáspár [3] adopted the so-called PL algorithm (described by Allgower and Georg [1]) to the given task. The evaluation of the functions f_i requires in general the forward integration of an ordinary differential equation (ODE). In this section we describe briefly a generalized version of the algorithm.

In the first step we construct an orthogonal grid with grid sizes $\Delta_i (i = 1, 2, \dots, n)$ in the n -dimensional space. This grid subdivides the phase space into finite orthogonal domains to which we will refer as "cubes" for the sake of simplicity. Each cube has 2^n vertices and can be subdivided further into $n!$ simplices. Each of these simplices has $n+1$ vertices. (If $n = 2$ then the simplex is a triangle, if $n = 3$ then it is a tetrahedron.) The functions f_i can be now linearly interpolated on the simplectic grid, the linearized functions will be denoted by f_i^L . The solution of the linear equation system

$$f_i^L(x_1, x_2, \dots, x_n) = 0 \quad (i = 1, 2, \dots, n-1) \quad (2)$$

yields an equation of a straight line. If this line goes through the investigated simplex, then the line has two intersection points with the surface of this simplex and the inner part of the line represents a segment of the result in the n dimensional space.

This operation has to be repeated for each simplex in the investigated phase space. Since the number of simplices can be very large and the operation on each simplex is identical, this offers an ideal task for PVM [4]. The implementation is described in the next section.

3 The Parallel Algorithm

A simple problem involves 3-5 dimensional phase space and the complexity of the problem grows exponentially with the number of dimensions. In order to solve the equation system with prescribed precision we have to choose sufficiently small grid-size (Δ_i). Supposing that the number of points on each coordinate axis is N and the number of dimensions is n , the numbers of points where we have to evaluate each function will be N^n . This means that the CPU and memory requirements of the algorithm grow exponentially with the number of dimensions.

To design the parallel algorithm we have examined the simplex algorithm. The main statements are:

- a. Every simplex is independent from the results of the calculations in the other simplices.
- b. Since adjacent simplices have common vertices, the function values should not be re-computed for each simplex.
- c. We assume that the computation time for the the functions f_i is not negligible and could be different at different points.
- d. We only expect solution points in a few simplices and most of the simplices do not contain any solution points. (In the limit where the size of the simplices goes to zero we expect solution points "almost nowhere", on a subset of measure zero.)

Considering the first statement, i.e. that the computation in every simplex is independent suggests that the simplex could be the element of the parallelization, however, the computation steps could be also parallelized in a simplex e.g. by solving the linearized functions and the n different equations of the simplex facets in parallel method. The main disadvantage of the parallelization inside the simplex is that the cost of the communication between two processes in PVM is very high.

The implemented parallel program is based on a master-slave structure, where the master program distributes the phase space to smaller pieces (domains) and the slaves figure out the equation system in these domains. The major functions of the master program:

- reading the configuration files
- starting and stopping the slaves
- collecting the results from slaves
- load-balancing
- doing checkpoint restart

Files are handled only by the master program thus the slaves never compete for accessing files in the NFS server. The slave program essentially contains the serial version of the described Simplex Algorithm and solves the equations in the domain given by the master. The values of the functions are computed once by the slave, when the slave gets a new domain from the master program. In this manner the function values are multiply computed only on the boundary points between the domains. To minimize the number of boundary points the master program tries to create domains with approximately equal orthogonal sizes.

The load-balancing is provided by the master, because the phase space is divided into more domains than the number of processors. When the computation in any domain has been finished, the master sends the next domain to the next free slave. In this way the faster processors will get more jobs than the slower processors.

4 Recursive version of the PSA

The original equation (1) may contain parameters C_i besides the variables x_i . In this case solution sets emerge as multi-dimensional manifolds rather than 1D-lines. In many applications special, 1D subsets of these multidimensional manifolds are of real interest. We embedded the PSA into a recursive scheme, capable to compute these special lines directly.

The simplest case of such a recursion (Depth 1) is when there is one parameter: C_1 . In this case one might ask, how do the *extremal points* on the solution branches (1D lines) of the *original problem* vary as we change C_1 . The "dumb" approach to this question would be to let the PSA solve the original problem for many values of C_1 , select extrema on all diagrams and then connect them. The recursive approach is rather different.

It regards an extended, $n + 1$ dimensional problem. In order to isolate 1D branches, one needs one additional function/constraint: this is delivered by the condition that we are looking for extrema. While the first $n - 1$ functions and their evaluation remains identical to the original problem, the evaluation of the last, added function (extremal condition) requires the solution of an n -dimensional problem (in a very small domain).

We will now generalize the recursion to arbitrary depth. The next sentences might sound complicated, but this is as close as plain English can get to a recursive algorithm:

In a similar manner, now we could ask about how the extremal points of the branches of the $n + 1$ -dimensional problem change as we vary a new parameter C_2 . This would require the solution of an $n + 2$ dimensional problem, (the new dimension is spanned by C_2). In this $n + 2$ dimensional problem we have $n + 1$ functions, out of which $n - 1$ are "normal" functions, requiring integration of ODEs. There are two special functions: one is an n -dimensional problem, the second is an $n + 1$ dimensional problem, each to be solved by the PSA (in a small domain, though). The last mentioned $n + 1$ -dimensional problem has n

functions, out of which $n - 1$ are "normal", one is an n -dimensional problem. This leads to the general idea of a recursive scheme.

The practical application of this results is very wide. Even the simplest, depth-1 recursion can compute *stability boundaries* globally, inaccessible by any other method, to the best of our knowledge.

We have modified our PSA implementation for this requirement, resulting in a recursive algorithm, which is a substantial generalization of the original PSA. The key idea is simple (described here for depth-1 recursion):

- enlarge our equation system (1) with a new function which yields the required properties (eg. turning points, local extrema) at the roots of original equation system
- set the C_1 as a new variable
- solve this enlarged equation system using Simplex Algorithm described in section 2.

For example let us regard a simple 2D problem described by the next equation:

$$f_1(C_1, x_1, x_2) = \cos(x_1 + C_1) + C_1 \sin(C_1^2) - x_2 = 0 \quad (3)$$

where x_1 and x_2 are variables and C_1 is a constant parameter. (This is not a real technical problem, but simple enough for presentation.) Based on equation (3) we can express the solution of $f_1 = a$ as $x_2 = h_a(C_1, x_1)$, where $h(C_1, x_1, a) = \cos(x_1 + C_1) + C_1 \sin(C_1^2) - a$, shown in a small domain for $a = 0$ in figure 1. If

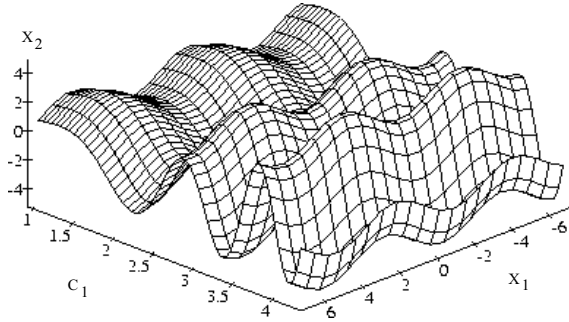


Fig. 1. $h = \cos(x_1 + C_1) + C_1 \sin(C_1^2)$

we want to get the turning points of the solution lines depending on C_1 , we have to extend equation (3) with a second equation $f_2(C_1, x_1, x_2) = 0$, identifying the turning points of h :

$$f_2(C_1, x_1, x_2) = \frac{\partial h(u, v, f_1(C_1, x_1, x_2))}{\partial v} = 0. \quad (4)$$

If we regard C_1 as a new variable, we have 3 variables and 2 functions (3) and (4), resulting in a "normal" 3D problem to be resolved by the PSA.

We remark that in most technical problems the solution function (here h) is not known in explicit form, so, in order to evaluate f_2 at a given point, the n -dimensional PSA has to be called as a subroutine (in a small domain). Since the complete problem is formulated as an $n + 1$ -dimensional PSA, and calls an n -dimensional PSA, this is a recursion scheme which can be extended (and has been implemented) to arbitrary depth.

In figure 2 and 3 we present the results of 3D recursive PSA for our example above. The first shown on a the $[C_1, x_1]$ plane and the second on the $[C_1, x_2]$ plane. We remark that f_2 may contain virtually *any* condition on the

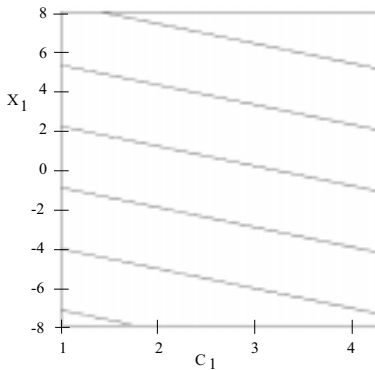


Fig. 2. Lines of extrema projected onto the $[C_1, x_1]$ plane

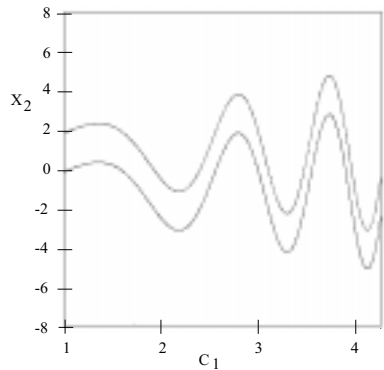


Fig. 3. Lines of extrema projected onto the $[C_1, x_2]$ plane

solution branch of the original, n -dimensional problem, including higher derivatives, singularities, etc. We emphasize that we did not restrict the dimension or recursion depth of the algorithm, so our code is capable of solving a large variety of problems emerging in applications.

5 Results and conclusions

Our parallel code has been applied to different technical problems in the past 4 years. The code was tested on many architectures running PVM. We have used it on MP, SMP platforms and on heterogeneous workstation clusters as well. Because of the domain partitioning and due to independence of the domains the speed-up is almost linear with the number of processors. The slave processes have measured the elapsed times (*user*, *system*, *real*) and the number of processed subdomains. This values were gathered by the master program and it was reported at the end of the runs. Table 5 shows the measured values and the *relative speed-up* (compared to the result of 30 processors) on an IBM SP2 computer, running with 30, 60 and 120 nodes. Considering the minimum and maximum numbers of processed subdomains, in our test case the load was quite

Table 1. The elapsed times depending on number of processors

Processor no	User time[s]		Real time[s]		Subdomains		Rel. Speed-up
	Max.	Avr.	Max.	Avr.	Min.	Max.	
30	26330	26030	26517	26226	55	56	1
60	13131	12794	13235	12893	27	28	2.00
120	6617	6345	6663	6392	13	14	3.98

well balanced between the processors. The PSA has been applied to a relatively wide range of problems in mechanics, we are giving here a very brief account of those:

- The global behavior of elastic rods with various end conditions is investigated in [7]. Here it is shown that in the well-known problem of a clamped-clamped beam there exist a nontrivial, tertiary equilibrium path connecting the already known first and second modes. This result was first shown by using the PSA.
- The equilibria of unilaterally constrained elastic fibers (as models for non-woven fabrics) is described in [6]. The paper relies heavily on numerical results which were obtained using the PSA in 3,4,5 and 7 dimensional spaces. Several *disconnected* equilibrium paths have been identified, inaccessible by other computation methods.
- Currently the equilibria of liquid bridges are being investigated by means of the PSA ([8]). This problem offers multiple challenges: although the basic bifurcation diagram is just 2-dimensional, solutions live on several, disconnected surfaces. This was the first real-life problem where the recursive version of the PSA was tested: we computed global stability boundaries for the liquid bridges in large parameter domains. The computations agree very well with earlier work, performed in smaller domains and more modest numerical tools.

The PSA proved to be a very useful tool in the global numerical investigation of low-dimensional BVPs related to ODEs. The maximal dimension investigated up to now is 7. The numerical results of the PSA could (and have been) used as starting data for traditional path-continuation codes. The embedding of the PSA into a recursive scheme is a substantial generalization. The new algorithm is (naturally) capable to perform everything that the old one did. In addition, it can compute a much wider class of problems, including global stability boundaries. The new algorithm has been successfully tested in finding the stability boundaries for liquid bridge equilibria.

Future tasks involve automated handling of numerically irrelevant, spurious solutions arising from discretization, automated mesh-refinement procedures and inter-active visualization. Several of these projects are under way.

Acknowledgements

This research was supported by OTKA grant F021307 and the USA-Hungarian Joint Fund Grant 656/96.

References

1. Allgower, E.L., Georg, K. *Numerical continuation methods: an introduction* Springer-Verlag, Berlin, New York, 1990.
2. Bieberbach L. *Differentialgleichungen* Springer, Berlin 1923
3. Domokos G., Gáspár Zs. *A Global Direct Algorithm for Path-Following and Active Static Control of Elastic Bar Structures* J. of Structures and Machines 23 (4), pp 549-571 (1995)
4. Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R., Sunderam V. *PVM 3 User's Guide and Reference Manual* Oak Ridge National Laboratory, Oak Ridge TN 37831-6367 Technical Report, ORNL/TM-12187, May, 108pp., 1993.
5. Szeberényi I., Domokos G., Gáspár Zs. *Solving structural mechanics problems by PVM and the Simplex Algorithm* Proc. 2nd EuroPVM'95, eds: Dongarra J. et al., Hermes, Paris, 1995., pp 247-252
6. Holmes P., Domokos G., Schmitt J., Szeberényi I. *Constrained Euler Buckling: an Interplay of Computation and Analysis* Comp. Meth. in Appl. Mech. and Eng. 170 (3-4) 175-207 (1999.)
7. Domokos G. *Global Description of Elastic Bars* ZAMM 74 (4) T289-T291
8. Domokos G., Steen P., Szeberényi I. *Simultaneously resolved bifurcation diagrams: a novel global approach to liquid figures of equilibrium* manuscript

Hyper-Rectangle Distribution Algorithm for Parallel Multidimensional Numerical Integration

Raimondas Čiegis^{1,2}, Ramūnas Šablinskas^{2,3}, and Jerzy Waśniewski⁴

¹ Institute of Mathematics and Informatics, Akademijos 4, LT-2600 Vilnius, Lithuania

`rc@fm.vtu.lt`

² Vilnius Gediminas Technical University, Saulėtekio 11, LT-2054, Vilnius, Lithuania

³ Vytautas Magnus University, Vileikos 8, 3035 Kaunas, Lithuania

`ramas@vdu.lt`

⁴ The Danish Computing Centre for Research and Education

UNI-C, Bldg. 305, DK-2800 Lyngby, Denmark

`Jerzy.Wasniewski@uni-c.dk`

Abstract. In this paper we consider parallel numerical integration algorithms for multi-dimensional integrals. A modified algorithm of hyper-rectangle selection and distribution strategy is proposed for the implementation of globally adaptive parallel quadrature algorithms. A list of subproblems is distributed among slave processors. Numerical results on the *SP2* computer and on a cluster of workstations are reported. A test problem where the integrand function has a strong corner singularity is investigated.

Keywords: parallel adaptive integration, distributed-memory parallel computers, load-balancing, redistribution of tasks.

1 Introduction

The problem considered in this paper is the numerical approximation of the multi-dimensional integral

$$I(f, \Omega) = \int_{\Omega} f(x) dx \quad (1)$$

to some given accuracy ε , where $\Omega = [a_1, b_1] \times [a_2, b_2] \cdots [a_n, b_n]$ is the range of integration and $f(x)$ is the integrand function. We are interested in implementing globally adaptive integration algorithms on distributed memory parallel computers. The algorithms of this paper are targeted at clusters of workstations with standard message passing interface. We use PVM in our numerical experiments.

Numerical integration algorithms attempt to approximate $I(f, \Omega)$ by the sum

$$S_N(f) = \sum_{i=1}^N w_i f(x_i). \quad (2)$$

The numbers w_i are called *weights*, and the x_i are called *knots*. Efficient numerical algorithms are adaptive and some strategy for selection of hyper-rectangles for further subdivision is included into the algorithm. We use the cubature rule pair, which was proposed by Genz and Malik [1980]. Parallel adaptive algorithms were investigated in many papers [1995, 1996, 1992]. Mostly these algorithms are targeted at shared memory computers.

In our previous paper [1998] we considered one parallel numerical integration algorithm for multidimensional integrals. It is targeted for a numerical approximation of integrals when the integrand function f has strong singularities. The load balancing of the static subdivision – dynamic distribution algorithm becomes very poor and the quality of the parallel adaptive algorithm also degrades seriously (see [1997]). By using a new hyper-rectangle selection algorithm the quality of the parallel adaptive cubature method is improved considerably, but additional costs of communications are also increased since a more fine-grain subdivision of the integration region Ω is implemented. This parallel algorithm attempts to calculate the multi-dimensional integral faster when we use p processors. The second important goal in parallel computing is to solve a larger problem. In our case the more accurate approximation of the integral requires to use a larger list of subproblems. Hence we propose a parallel integration algorithm in which the list of subproblems is distributed among processors.

We summarize the remainder of this paper. In Section 2 we describe the hyper-rectangle selection algorithm and present results of extensive numerical experiments on a *MIMD* computer and a cluster of workstations. The new master–slave algorithm with the distributed list of tasks is described in Section 3.

2 Hyper-Rectangle Selection Strategy

We describe very briefly a parallel algorithm for multi-dimensional cubature. The numerical integration problem can be considered as a dynamic resource problem, since the number of subdivisions n_j required for the numerical approximation of the integral $I(f, \Omega_j)$ in subregions Ω_j can be very different (see [1998]). If our parallel computer is a cluster of multi-user workstations then the available processing capacity per node may change during computations, hence we have a dynamic resource computer. The load balancing problem can be considered as the mapping of a dynamic resource problem onto a dynamic resource computer. The algorithm must include some mechanism for the migration of subproblems from over-loaded processes to under-loaded processes at run-time. The elementary problem in the task list is estimation of the integral $I(f, P_j)$, where P_j is a hyper-rectangle obtained by the Genz-Malik algorithm.

We use the following hyper-rectangle selection algorithm, which is based on the well known parallel algorithm prototype: master–slave algorithm.

Hyper-Rectangle Selection Algorithm (HRS)

The *master* part of the algorithm is given by the following code.

- 1 Form the task-list of hyper-rectangles P_j , $j = 1, 2, \dots, s$, ranked by error estimates so that $\varepsilon_1 < \varepsilon_2 < \dots < \varepsilon_s$.
- 2 Send the initial data portions to slaves.
- 3 **do while** (ε is greater than required accuracy ε_0)
 - receive from a slave the ranked list of R_{recv} hyper-rectangles;
 - put these results to the main ranked list of hyper-rectangles P ;
 - update the integral approximation and error estimates ε ;
 - send to the slave R_{send} hyper-rectangles with the largest errors.
- end do**

The *slave* part of the algorithm is given by the following code.

- 1 Receive a job list from the master process.
- 2 Apply the adaptive numerical integration algorithm and make $R_{recv} - R_{send}$ subdivisions.
- 3 Rank the obtained list of hyper-rectangles according to their error estimates and send the result to the master process.

The algorithm contains two free parameters R_{send} and R_{recv} . The optimal values of these parameters depend on the dimension of the integral and on the ratio between computation and communication rates of a parallel computer. The *quick-sort* method is used by slave processes to rank the list of subproblems.

We investigate the performance of the proposed hyper-rectangle selection algorithm. Let consider the following test problem (see [1995])

$$\int_0^1 \int_0^1 \int_0^1 (x_1 x_2 x_3)^{-0.9} dx_1 dx_2 dx_3, \quad (3)$$

where the integrand function has a strong corner singularity. We compute a numerical approximation of the integral (3) to the relative accuracy $\varepsilon = 0.002$. The *SP2* computer and a cluster of *RS6000* workstations were used in computations.

We define the speed-up S_p and the efficiency E_p as

$$S_p = \frac{T_s}{T_p}, \quad E_p = \frac{S_p}{p}, \quad (4)$$

where T_p is the time used to solve the given problem on p processors, T_s represents the time for the sequential globally adaptive algorithm.

Table 1 presents the numbers of subdivisions N_p , timings T_p in seconds, speed-ups S_p and the efficiency E_p on the *SP2* computer. We set the following values of parameters $R_{send} = 200$, $R_{recv} = 700$.

Table 2 gives analogous information for the cluster of workstations *RS6000*. We wish to test the quality of the proposed algorithm, therefore we choose to add appropriate delays in the evaluation of integrand function. A dummy loop is incorporated into the code and it is repeated 5 times.

Table 1. Numerical results on the *SP2* computer

p	Number of subdiv. N_p	Time T_p	Speed-up S_p	Efficiency E_p
1	217698	77.9	0.957	0.957
2	217897	39.8	1.874	0.937
4	219295	20.8	3.586	0.896
7	220392	12.7	5.873	0.839
15	226984	7.4	10.080	0.672

Table 2. Numerical results on the cluster of workstations

p	Number of subdiv. N_p	Time T_p	Speed-up S_p	Efficiency E_p
1	217698	313.3	0.945	0.945
2	217897	167.4	1.768	0.884
3	218596	117.7	2.515	0.838
4	219295	92.0	3.217	0.804
5	219494	74.5	3.973	0.795
6	220193	64.0	4.625	0.771
7	220392	57.8	5.121	0.732

From the the results given in Tables 1,2 we see that the quality of the proposed algorithm degrades only slightly when the number of processors is increased.

Next we investigated the sensitivity of the algorithm to the selection of parameters R_{send} and R_{recv} . Table 3 presents the dependency of computation time and computation efficiency from parameters R_{send} and R_{recv} . The computations were carried out on the cluster of five *RS6000* workstations. A dummy loop is again incorporated into the code. Numerical experiments show that the algorithm is not very sensitive to the selection of these parameters and the time of computations varies by about ten percents for a large range of values of R_{send} and R_{recv} .

Table 3. The influence of R_{send} and R_{recv} to the computation time T_p

R_{send}	R_{recv}	Number of subdivisions N_p	Time T_p	Speed-up S_p
10	20	217204	281.7	1.051
10	100	218924	77.2	3.835
50	300	218744	72.6	4.078
50	700	223194	70.8	4.182
200	700	219494	74.5	3.973
400	2000	224394	74.7	3.963

Table 4. Results of the experiments without additional delay

R_{send}	R_{recv}	Number of subdivisions N_p	Time T_p	Speed-up S_p
10	50	217555	49.6	1.45
100	300	217795	35.4	2.03
100	900	221195	29.3	2.46
200	400	217795	43.9	1.64
200	1000	219995	31.3	2.30

As introduced in [1991], the time of point-to-point communication between two processes can be estimated as

$$T_{send} = \alpha + \beta n, \quad (5)$$

where α is the latency (or start-up time) and β is the incremental time per data items moved; its reciprocal is called bandwidth. The values of these parameters are $\alpha = 2ms$, $\beta = 1s/MB$ for the public domain version of PVM using Ethernet, while PVMe on SP2 achieves $\alpha = 0.1ms$, $\beta = 0.03s/MB$.

Table 4 presents the results of experiments carried out on the cluster of four RS6000 workstations when no additional delay in integral function is used. We still obtain a speed-up of the parallel algorithm.

Table 5. The influence of problem complexity to the efficiency of the computations

ε	Number of subdivisions N_p	Time T_p	Speed-up S_p
2.0	226984	7.4	10.080
1.0	332484	11.0	10.855
0.5	503984	15.3	11.774

We also investigated the efficiency of the parallel integration algorithm when the difficulty of the problem was continuously increased. Table 5 gives the results obtained on the SP2 computer with 16 processors for different values of the accuracy parameter ε . We note that only one process per node can be run on SP2, hence one master process and 15 slave processes were used in computations.

The analysis of profiles shows that the part of the master algorithm for keeping ranked lists of hyper-rectangles P_j becomes critical operation for large lists. Hence an additional small list is introduced for keeping results from slaves. Due to this modification the master reduces a frequency of searching and updating the main global list of hyper-rectangles. In Table 6 we present timings T_4 for different lengths L of an additional list of hyper-rectangles. The computations were carried out on the cluster of four RS600 workstations with $R_{send} = 100$ and $R_{recv} = 900$. All previous computations were carried out with $L = 2000$.

Table 6. The influence of L to the computation time

R_{send}	R_{recv}	Length L	Time T_4
100	900	0	45.3
100	900	1000	45.3
100	900	2000	37.1
100	900	4000	34.2
100	900	8000	31.1

3 The Parallel Algorithm with Distributed List of Tasks

So far we have been interested in the solution of the integration problem as fast as it was possible. The parallel algorithms also enable us to solve larger problems as well. The HRS algorithm was organized in such a way, that the task list was maintained by the master process. Therefore the available master host memory defines the size of the task list. The task list size determines the maximum accuracy ε we can reach. The HRS algorithm does not utilize the memory of the slave processors. We propose a modified algorithm which utilizes the slave memory. Hence we can solve the larger problem than it is possible to solve with one computer. This is our primary goal. At the same time we demand that the quality and efficiency of the new algorithm must not degrade dramatically.

The Algorithm with Distributed List of Tasks (DLT)

The *slave* part of the algorithm is given by the following code.

1. Receive a job list from the master process.
2. Join the received job list with the local job list.
3. Take $R_{recv} + R_{local} - R_{send}$ jobs with the largest error estimates from the local job list and then apply quadrature $R_{recv} + R_{local} - R_{send}$ times.
4. Send the R_{recv} jobs with the largest errors to the master process.
5. Put the remaining jobs from the result list into the local job list.

The *master* part of the algorithm is given by the following code.

1. Get the information from slaves about the maximal sizes of the local job lists $M_k, k = 1, 2, \dots, p$.
2. Form the initial task-list of hyper-rectangles $P_j, j = 1, 2, \dots, s$, ranked by error estimates so that $\varepsilon_1 < \varepsilon_2 < \dots < \varepsilon_s$.
3. Send the initial data portions to slaves.

```

4. do while ( $\varepsilon$  is greater than required accuracy  $\varepsilon_0$ )
    if (the master has a sufficient number of free memory blocks)
        a) receive from the  $j$ -th slave the ranked list of  $R_{recv}$ 
           hyper-rectangles, the integral approximation value  $S_j$  and
           the estimated error of all hyper-rectangles kept in its
           local list  $err_j$ ,
        b) put the results into the main ranked list  $P$ ,
        c) update the integral approximation  $S$  and error estimate  $\varepsilon$ ,
        d) send to the  $j$ -th slave  $R_{send}$  hyper-rectangles with the
           largest errors,
        e) update the  $S_j$  and  $err_j$ .
    else
        f) find a set of slaves  $J$  which have free memory blocks:
            $J = \{j | M_j > 0\}$ ,
        g) define the numbers of tasks  $R_{distr}(j), j \in J$  that must
           be distributed among the slaves,
        h) for( $j \in J$ )
            implement steps a,b and c,
            send to the  $j$ -th slave  $R_{send}$  hyper-rectangles with the
            largest errors and  $R_{distr}(j)$  hyper-rectangles
            with the smallest errors,
            update the  $S_j, err_j$  and  $M_j$ .
        end for
    end if
end do

```

The algorithm contains three parameters R_{send} , R_{recv} and R_{local} . The optimal values of these parameters depend on the dimension of the integral and on the ratio between computation and communication rates of a parallel computer. If we assume that the size of slave's local task list is equal to zero, we will have the HRS algorithm. On the other hand, if the memory allocation condition in the step 3 of the master algorithm does not fail and $R_{local} = 0$, we will also have the HRS algorithm.

Table 7 presents the results of computational experiments carried out on the cluster of RS600 workstations. Here we set $R_{send} = 100$, fix the number of adaptive subdivisions to 800, and take different values of R_{local} . Comparing the results from Tables 4, 7 we deduce that the DLT algorithm has the same efficiency as the HRS algorithm. We also note that the increase of R_{local} reduces the communication time and introduces parallelism in the list updating step.

The DLT algorithm also enables us to solve the bigger problems by utilizing the memory resources of the slave processes. Suppose the slave process can fit 250000 records of the hyper-rectangles in the local list memory. Table 8 shows the accuracy of the numerical approximation achieved for various number of processes.

Table 7. The influence of R_{recv} and R_{local} to the computation times

p	R_{send}	R_{recv}	R_{local}	Time T_p
4	100	900	0	44.4
4	100	800	100	39.1
4	100	700	200	38.0
4	100	600	300	33.3
4	100	500	400	32.2
4	100	400	500	27.7

Table 8. The accuracy achieved with DLT algorithm as to compared to HRS algorithm

p	Accuracy achieved ε	Time T_p	Total hyper-rectangles
1	0.493	258.4	500000
2	0.251	181.3	750000
3	0.152	183.0	1000000
4	0.096	184.2	1250000
5	0.069	193.2	1500000
6	0.049	235.1	1750000

References

[1995] Bull, J.M., Freeman, T.L.: Parallel Globally Adaptive Algorithms for Multi-dimensional Integration. *Applied Numerical Mathematics* **19** (1995) 3–16

[1991] Freeman T.L. and Phillips C.: *Parallel Numerical Algorithms*. Prentice Hall, New York, London, Toronto, Sydney, Tokyo, Singapore (1991).

[1997] Čiegis, R., Šablinskas, R., Waśniewski, J.: Numerical Integration on Distributed Memory Parallel Systems. In: Bubak, M., Dongarra, J., Waśniewski, J. (eds.): *Recent Advances in PVM and MPI. Lecture Notes in Computer Science*, Vol. 1332. Springer-Verlag, Berlin Heidelberg New York (1997) 329–336

[1998] Čiegis, R., Šablinskas, R., Waśniewski J.: Hyper-rectangle selection strategy for parallel adaptive numerical integration. In: B.Kagstrom, Dongarra, J., Elmroth, E., Waśniewski, J. (eds.): *Proc. 4th International Workshop PARA98. Lecture Notes in Computer Science*, Vol. 1541. Springer-Verlag, Berlin Heidelberg New York (1998) 71–75

[1980] Genz, A.C., Malik, A.A.: Remarks on Algorithm 006: an Adaptive Algorithm for Numerical Integration Over an N-dimensional Rectangular Region. *J. Comput. Appl. Math.* **6** (1980) 295–302

[1996] Keister, B.D.: Multi-dimensional Quadrature Algorithms. *Computers in Physics*, **10** (1996) 119–122

[1992] Miller, V.A., Davis, G.J.: Adaptive Quadrature on a Message Passing Multiprocessor. *J. Parallel Distributed Comput.*, **14** (1992) 417–425

Parallel Monte Carlo Algorithms for Sparse SLAE Using MPI

V. Alexandrov¹ and A. Karaivanova²

¹ Department of Computer Science, University of Liverpool
Chadwick Building, Peach Street, Liverpool, L69 7ZF, UK
`vassil@csc.liv.ac.uk`

² Central Laboratory for Parallel Processing, Bulgarian Academy of Sciences
Acad. G. Bonchev St., bl. 25 A, 1113, Sofia, Bulgaria
`anet@copern.acad.bg`

Abstract. The problem of solving sparse Systems of Linear Algebraic Equations (SLAE) by parallel Monte Carlo numerical methods is considered. The almost optimal Monte Carlo algorithms are presented. In case when a copy of the non-zero matrix elements is sent to each processor the execution time for solving SLAE by Monte Carlo on p processors is bounded by $O(nNdT/p)$ where N is the number of chains, T is the length of the chain in the stochastic process, which are independent of matrix size n , and d is the average number of non-zero elements in the row. Finding a component of the solution vector requires $O(NdT/p)$ time on p processors, which is independent of the matrix size n .

1 Introduction

It is known that Monte Carlo methods give statistical estimates for the components of the solution vector of SLAE by performing random sampling of a certain random variable whose mathematical expectation is the desired solution [11,12]. We consider Monte Carlo methods for solving SLAE since: **firstly**, only $O(NT)$ steps are required to find an element of the inverse matrix (MI) or component of the solution vector of SLAE (N is a number of chains and T is a measure on the chains length in the stochastic process, which are independent of n) and **secondly**, the sampling process for stochastic methods is inherently parallel. In comparison, the direct methods of solution require $O(n^3)$ sequential steps for dense matrices when the usual elimination or annihilation schemes (e.g non-pivoting Gaussian Elimination, Gauss-Jordan methods) are employed [4]. While considering general sparse matrices a reordering algorithms are usually used before applying direct or iterative methods of solution. The time required for reordering is $O(n^2)$ for straightforward reordering or $O(Z \log(n))$ if binary trees are used [13], where Z is the number of non-zero elements in the matrix.

Consequently the computation time for very large problems or for real-time problems can be prohibitive and prevents the use of many established algorithms. Therefore due to their properties, their inherent parallelism and loose

data dependencies Monte Carlo algorithms can be implemented on parallel machines very efficiently and thus may enable us to solve large-scale problems which are sometimes difficult or prohibitive to be solved by the well-known numerical methods.

Generally three Monte Carlo methods for Matrix Inversion (MI) and finding a solution vector of System of Linear Algebraic Equations (SLAE) can be outlined: with absorption, without absorption with uniform transition frequency function, and without absorption with almost optimal transition frequency function.

In the case of **fine grained** setting, recently Alexandrov, Megson and Dimov have shown that an $n \times n$ matrix can be inverted in $3n/2 + N + T$ steps on regular array with $O(n^2NT)$ cells [10]. Alexandrov and Megson have also shown that a solution vector of SLAE can be found in $n + N + T$ steps on regular array with the same number of cells [2]. A number of bounds on N and T have been established, which show that these designs are faster than the existing designs for large values of n [10,2].

The **coarse grained** case for MI is considered in [1]. The **coarse grained** parallel Monte Carlo algorithms for solving dense SLAE and finding a dominant eigenvalue are considered in [3] and [6] respectively. In this paper we extend this implementation approach for sparse SLAE in MIMD environment, i.e. a cluster of workstations under MPI in our case. We also derive an estimate on time complexity using CGM model.

The **Coarse Grained Multicomputer** model, or $CGM(n, p)$ for short, which is the architectural model to be used in this paper is a set of p processors with $O(\frac{n}{p})$ local memory each, connected to some arbitrary interconnection network or a shared memory. The term “**coarse grained**” refers to the fact that (as in practice) the size $O(\frac{n}{p})$ of each local memory is defined to be “considerably larger” than $O(1)$. Our definition of “considerably larger” will be that $\frac{n}{p} \geq p$. This is clearly true for all currently available coarse grained parallel machines. For determining time complexities we will consider both, local computation time and inter-processor communication time, in the standard way.

For parallel algorithms for SLAE to be relevant in practice, such algorithms must be **scalable**, that is, they must be applicable and efficient for a wide range of ratios $\frac{n}{p}$. The use of CGM helps to ensure that the parallel algorithms designed are not only efficient in theory, but also they result in efficient parallel software with fast running time on real data. Experiments have shown that in addition to the scalability, the CGM algorithms typically quickly reach the point of optimal speedup for reasonable data sets. Even with modest programming efforts the actual results obtained for other application areas have been excellent [5].

In this paper we focus mainly on the case when a copy of the non-zero elements of a sparse matrix is sent to each processor.

2 Stochastic Methods and SLAE

Assume that the system of linear algebraic equations (SLAE) is presented in the form:

$$x = Ax + \varphi \quad (1)$$

where A is a real square $n \times n$ matrix, $x = (x_1, x_2, \dots, x_n)^t$ is a $1 \times n$ solution vector and $\varphi = (\varphi_1, \varphi_2, \dots, \varphi_n)^t$ is a given vector. (If we consider the system $Lx = b$, then it is possible to choose non-singular matrix M such that $ML = I - A$ and $Mb = \varphi$, and so $Lx = b$ can be presented as $x = Ax + \varphi$.) Assume that A satisfies the condition $\max_{1 \leq i \leq n} \sum_{j=1}^n |a_{ij}| < 1$, which implies that all the eigenvalues of A lie in the unit circle. The matrix and vector norms are determined as follows: $\|A\| = \max_{1 \leq i \leq n} \sum_{j=1}^n |a_{ij}|$, $\|\varphi\| = \max_{1 \leq i \leq n} |\varphi_i|$.

Suppose that we have Markov chains with n - states. The random trajectory (chain) T_i of length i starting in state k_0 is defined as $k_0 \rightarrow k_1 \rightarrow \dots \rightarrow k_j \rightarrow \dots \rightarrow k_i$ where k_j is the number of the state chosen, for $j = 1, 2, \dots, i$. The following probability definitions are also important: $P(k_0 = \alpha) = p_\alpha$, $P(k_j = \beta | k_{j-1} = \alpha) = p_{\alpha\beta}$ where p_α is the probability that the chain starts in state α and $p_{\alpha\beta}$ is the transition probability to state β from state α . Probabilities $p_{\alpha\beta}$ define a transition matrix P . We require that $\sum_{\alpha=1}^n p_\alpha = 1$, $\sum_{\beta=1}^n p_{\alpha\beta} = 1$ for any $\alpha = 1, 2, \dots, n$, the distribution $(p_1, \dots, p_n)^t$ is acceptable to vector g and similarly the distribution $p_{\alpha\beta}$ is acceptable to A [11].

Consider the problem of evaluating the inner product of a given vector g with the vector solution of (1)

$$(g, x) = \sum_{\alpha=1}^n g_\alpha x_\alpha \quad (2)$$

It is known [11] that the mathematical expectation $E\Theta^*[g]$ of random variable $\Theta^*[g]$ is:

$$E\Theta^*[g] = (g, x)$$

$$\text{where } \Theta^*[g] = \frac{g_{k_0}}{p_{k_0}} \sum_{j=0}^{\infty} W_j \varphi_{k_j} \quad (3)$$

$$\text{and } W_0 = 1, \quad W_j = W_{j-1} \frac{a_{k_{j-1}k_j}}{p_{k_{j-1}k_j}}$$

We use the following notation for a partial sum (3) $\theta_i[g] = \frac{g_{k_0}}{p_{k_0}} \sum_{j=0}^i W_j \varphi_{k_j}$. According to the above conditions on the matrix A , the series $\sum_{j=0}^{\infty} W_j \varphi_{k_j}$ converges for any given vector φ and $E\theta_i[g]$ tends to (g, x) as $i \rightarrow \infty$. Thus $\theta_i[g]$ can be considered an estimate of (g, x) for i sufficiently large.

Now we define the Monte Carlo method. To find one component of the solution, for example the r -th component of x , we choose $g = e(r) = (0, \dots, 0, 1, 0, \dots, 0)$ where the one is in the r -th place. It follows that $(g, x) = \sum_{\alpha=1}^n e_\alpha(r) x_\alpha = x_r$ and the corresponding Monte Carlo method is given by

$$x_r \approx \frac{1}{N} \sum_{s=1}^N \theta_i[e(r)]_s \quad (4)$$

where N is the number of chains and $\theta_i[e(r)]_s$ is the value of $\theta_i[e(r)]$ in the s -th chain.

The **probable error** of the method, is defined as $r_N = 0.6745\sqrt{D\theta/N}$, where $P\{|\bar{\theta} - E(\theta)| < r_N\} \approx 1/2 \approx P\{|\bar{\theta} - E(\theta)| > r_N\}$, if we have N independent realizations of random variable (r.v.) θ with mathematical expectation $E\theta$ and average $\bar{\theta}$ [11].

It is clear from the formula for r_N that the number of chains N can be reduced by a suitable choice of the transition probabilities that reduces the variance for a given probable error. This idea leads to Monte Carlo methods with minimal probable error.

The key results concerning minimization of probable error and the definition of **almost** optimal transition frequency for Monte Carlo methods applied to the calculation of inner product via iterated functions are presented in [10]. According to [10,9] and the principal of collinearity of norms [10] we can choose $p_{\alpha\beta}$ proportional to the $|a_{\alpha\beta}|$.

In case of Almost Optimal Monte Carlo [1,2] the following transition probabilities $p_{\alpha\beta}$ are applied :

$$p_{\alpha\beta} = \frac{|a_{\alpha\beta}|}{\sum_{\beta} |a_{\alpha\beta}|} \text{ for } \alpha, \beta = 1, 2, \dots, n.$$

In case of $\|A\| \geq 1$ or very close to 1 we can use the Resolvent Monte Carlo method [7] to reduce the matrix norm and to speedup the computations.

3 Parameters Estimation and Discussion

We will outline the method of estimation of N and T in case of **Monte Carlo method without absorbing states** since it is known that these methods require less chains than the methods with absorption to reach the same precision [2,1]. In case of Monte Carlo with absorption the parameter estimation can be done in the same way. We will consider Monte Carlo methods with almost optimal (MAO) transition frequency function. We assume that the following conditions $\sum_{\beta=1}^n p_{\alpha\beta} = 1$ for any $\alpha = 1, 2, \dots, n$ must be satisfied and transition matrix P might have entries $p_{\alpha\beta} = \frac{|a_{\alpha\beta}|}{\sum_{\beta} |a_{\alpha\beta}|}$ for $\alpha, \beta = 1, 2, \dots, n$.

The estimator Θ^* for SLAE was defined as follows

$$\begin{aligned} E\Theta^*[g] &= (g, x), \\ \text{where } \Theta^*[g] &= \frac{g_{k_0}}{p_{k_0}} \sum_{j=0}^{\infty} W_j \varphi_{k_j} \\ \text{and } W_0 &= 1, \quad W_j = W_{j-1} \frac{a_{k_{j-1}k_j}}{p_{k_{j-1}k_j}}. \end{aligned} \tag{5}$$

The sum for Θ^* must be dropped when $|W_i \varphi_{k_i}| < \delta$ [11].

Note that

$$|W_i \varphi_{k_i}| = \left| \frac{a_{\alpha_0 \alpha_1} \cdots a_{\alpha_{i-1} \alpha_i}}{\|A\| \cdots \|A\|} \right| |\varphi_{k_i}| = \|A\|^i \|\varphi\| < \delta.$$

Then it follows that

$$T = i \leq \frac{\log(\delta/\|\varphi\|)}{\log\|A\|}.$$

It is easy to find [11] that $|\Theta^*| \leq \frac{\|\varphi\|}{(1-\|A\|)}$, which means that variance of r.v. Θ^* is bounded by its second moment: $D\Theta^* \leq E\Theta^{*2} = \frac{\|\varphi\|^2}{(1-\|A\|)^2} \leq \frac{f^2}{(1-\|A\|)^2}$. According to the Central Limit Theorem for the given error ϵ

$$N \geq \frac{0.6745^2 D\eta^*[g]}{\epsilon^2} \text{ and thus } N \geq \frac{0.6745^2}{\epsilon^2} \frac{f^2}{(1-\|A\|)^2} \quad (6)$$

is a lower bound on N which is independent of n .

It is clear that T and N depend only on the matrix norm and precision.

4 Parallel Implementation

We implement parallel Monte Carlo algorithms on a cluster of workstations under MPI. We assume virtual star topology and we apply master/slave approach.

Inherently, Monte Carlo methods for solving SLAE allow us to have minimal communication, i.e. to pass the non-zero elements of the sparse matrix A to every processor, to run the algorithm in parallel on each processor computing $\lceil n/p \rceil$ components of the solution vector and to collect the results from slaves at the end without any communication between sending non-zero elements of A and receiving partitions of x . Even in the case we compute only k components ($1 \leq k \leq n$) of the solution vector we can divide evenly the number of chains among the processors, e.g. distributing $\lceil kN/p \rceil$ chains on each processor. The only communication is at the beginning and at the end of the algorithm execution which allows us to obtain very high efficiency of parallel implementation. Therefore, by allocating the master in the central node of the star and the slaves in the remaining nodes, the communication is minimized.

Since we need to compute n components of the vector solution each requiring N chains of length T and having d non-zero elements in average in a row of a given sparse matrix on p processors in parallel, the time is $O(nNdT/p)$. This estimate includes logical operations, multiplications and additions.

5 Numerical Tests

The numerical tests are made on a cluster of 48 Hewlett Packard 900 series 700 Unix workstations under MPI (version 1.1). The workstations are networked via 10Mb switched ethernet segments and each workstation has at least 64Mb RAM and run at least 60 MIPS.

We have carried out several type of experiments. First, we have considered the case when one component of the solution vector is computed. In this case each processor executes the same program for N/p number of trajectories, i.e. it computes N/p independent realizations of the random variable. At the end the host processor collects the results of all realizations and computes the desired

value. The computational time does not include the time for initial loading of the matrix because we consider our problem as a part of bigger problem (for example, *spectral portraits of matrices*) and suppose that every processor constructs it.

Second we have considered computing an inner product (h, x) .

The **parallel efficiency** E is defined as:

$$E(X) = \frac{ET_1(X)}{pET_p(X)},$$

where X is a Monte Carlo algorithm, $ET_p(X)$ is the expected value of the computational time for implementation the algorithm X on a system of p processors.

Table 1. Implementation of the **Monte Carlo Algorithm** using MPI for calculating one component of the solution (number of trajectories - 100000).

	1pr.	2pr.	2pr.	4pr.	4pr.	5pr.	5pr.	8pr.	8pr.
	$T(ms)$	$T(ms)$	E	$T(ms)$	E	$T(ms)$	E	$T(ms)$	E
SLAE n = 128	38	19	1	15	0.63	12	0.63	8	0.6
SLAE n = 1024	28	14	1	8	0.9	6	0.93	4	0.9
SLAE n = 2000	23	11	1.04	6	0.96	5	0.92	5	0.6

Table 2. Implementation of the **Monte Carlo Algorithm** for evaluation of the scalar product (h, x) , where x is the unknown solution vector, and h is a given vector, using MPI (number of trajectories - 100000).

	1pr.	2pr.	2pr.	4pr.	4pr.	5pr.	5pr.	8pr.	8pr.
	$T(ms)$	$T(ms)$	E	$T(ms)$	E	$T(ms)$	E	$T(ms)$	E
SLAE n = 128	16	8	1	4	1	3	1.01	3	0.7
SLAE n = 1024	105	53	0.98	28	0.94	21	1	14	0.94
SLAE n = 2000	167	84	0.99	58	0.7	42	0.8	33	0.7

In all cases the test matrices are sparse and are stored in **packed row format** (i.e. only non-zero elements). The average number of non-zero elements per matrix row is $d = 52$ for $n = 128$, $d = 57$ for $n = 1024$ and $d = 56$ for $n = 2000$ respectively. The results for average time and efficiency are given in tables 1 and 2 and look promising. The relative accuracy is 10^{-3} .

As you can see in case when one component of the vector solution is computed the time does not depend on the matrix size n . When all the components of the solution vector are computed the time is linear of the matrix size n . This corroborates our theoretical results.

6 Conclusion

In our parallel implementation we have to compute n components of the solution vector of sparse SLAE in parallel. To compute a component of the solution vector we need N independent chains with length T for matrix with d non-zero elements per row in average, and for n components in parallel we need nN such independent chains of length T , where N and T are the mathematical expectations of the number of chains and chain length, respectively. So the execution time on p processors for solving SLAE by Monte Carlo is bounded by $O(nNdT/p)$ (excluding initialization communication time). According to the discussion and results above N and T depend only on the matrix norm and precision and do not depend on the matrix size. Therefore the Monte Carlo methods can be efficiently implemented on MIMD environment and in particular on a cluster of workstations under MPI.

In particular it should be noted that the Monte Carlo methods are well suited to large problems where other solution methods are impractical or impossible for computational reasons, for calculating quick rough estimate of the solution vector, and when only a few components of the solution vector are desired. This method also do not require any reordering strategies to be implemented. Consequently, if massive parallelism is available and if low precision is acceptable, Monte Carlo algorithms could become favorable for $n \gg N$.

References

1. Alexandrov, V., Lakka, S.: Comparison of three Parallel Monte Carlo Methods for Matrix Inversion, Proc. of EUROPAR96, Lyon, France, Vol II (1996), 72-80
2. Alexandrov, V., Megson, G.M.: Solving Sytem of Linear algebraic Equations by Monte Carlo Method on Regular Arrays, Proc. of PARCELLA96, 16-20 September, Berlin, Germany, (1996) 137-146
3. Alexandrov, V., Rau-Chaplin, A., Dehne, F., Taft, K.: Efficient Coarse Grain Monte Carlo Algorithms for Matrix Computations using PVM, LNCS 1497, Springer, August, (1998) 323-330
4. Bertsekas, D.P., Tsitsiklis : Parallel and Distributed Computation, Prentice Hall, (1989)
5. Dehne, F., Fabri, A., Rau-Chaplin, A.: Scalable parallel geometric algorithms for multicomputers, Proc. 7th ACM Symp. on Computational Geometry, (1993)

6. Dimov, I., Alexandrov, V., Karaivanova, A.: Implementation of Monte Carlo Algorithms for Eigenvalue Problem using MPI, LNCS 1497, Springer, August (1998) 346-353
7. Dimov, I., Alexandrov, V.: A New Highly Convergent Monte Carlo Method for Matrix Computations, *Mathematics and Computers in Simulation*, Vol. **47**, No 2-5, North-Holland, August (1998) 165-182
8. Golub, G.H., Ch., F., Van Loan: *Matrix Computations*, The Johns Hopkins Univ. Press, Baltimore and London, (1996)
9. Halton, J.H.: *Sequential Monte Carlo Techniques for the Solution of Linear Systems*, TR 92-033, University of North Carolina at Chapel Hill, Department of Computer Science, (1992)
10. Megson, G.M., Aleksandrov, V., Dimov, I.: Systolic Matrix Inversion Using Monte Carlo Method, *J. Parallel Algorithms and Applications*, Vol. **3**, (1994) 311-330
11. Sobol', I.M.: *Monte Carlo numerical methods*. Moscow, Nauka, (1973) (Russian)(English version Univ. of Chicago Press 1984).
12. Westlake J.R.: *A Handbook of Numerical Matrix Inversion and Solution of Linear Equations*, John Wiley and Sons, New York, (1968)
13. Gallivan, K., Hansen, P.C., Ostromsky, Tz., Zlatev, Z.: A Locally Optimized Re-ordering Algorithm and its Application to a Parallel Sparse Linear System Solver, *Computing* V. **54**, (1995) 39-67

A Method for Model Parameter Identification Using Parallel Genetic Algorithms

J.I. Hidalgo, M. Prieto, J. Lanchares and F.Tirado

B. de Andrés, S.Esteban and D. Rivera

Departamento de Arquitectura de Computadores y Automática,

Universidad Complutense de Madrid,

Av.Complutense s/n, E-20040 Madrid (Spain).

{hidalgo,mpmatias,julandan,ptirado}@dacya.ucm.es

{deandres,segundo}@dacya.ucm.es

Abstract. In this paper we present a new approach for the model parameters identification problem of real time systems based on parallel genetic algorithms. The method has been applied to an specific problem such as the control movement of a high speed ship where parallelization allows us to achieve the time constraints. The algorithm has been developed using MPI and the experimental results has been obtained on an SGI Origin 2000.

1 Introduction

Current industrial applications demand an increasing acceleration in many of their processes. In this kind of processes, real-time control is necessary in order to obtain appropriate responses. The mathematical modeling of these plants is usually not enough. Circumstances are variable and the parameters which define the model can vary during the process. The real-time identification of these parameters is a difficult problem, especially, when it is required in critical time. Examples of these applications are: the evolution of an economical model, pharmaceutical process control, rocket trajectory control ... Some classical methods such as the least square method [1] give good results for smooth plants but they usually need an appropriate initial solution. We present an alternative method based on genetic algorithms. The algorithm has been improved using a coarse grid parallelization in order to achieve time constraints [2]. The method has been tested using a real application where time is a critical factor: the heave movement control of a high speed ship.

The paper is organized as follows. Section 2 explains the problem formulation. In section 3, we introduce parallel genetic algorithm, their codification and their parameters. In section 4 the experimental results are explained and finally, in section 5, we obtain the conclusions of this work and some future works are proposed.

2 Problem Formulation

High speed ship control requires that in a very short time (in the order of seconds) the computer that controls the system could use an accurate mathematical model in order to obtain the best values of the variables which define the ship movements. These movements depends not only on its own features (length, mass, speed, ..) but also, on the particular characteristics of the sea (frequency, waves height, waves slope...). They can be considered as the combination of a set of more elemental motions called heave, pitch, roll, surge, sway, and yaw. We have focused on the heave movement, the vertical movement of its gravity center when it is attacked by a set of waves. A more detailed explanation of the others can be found on [3]. Thus, modeling a ship navigation consist of modeling each of its components in order to take into account their interactions and their common features. The influence of these components on the global ship movement depends not only on the ship characteristics and its speed but also on the sea behavior.

Different ships models have been proposed on the literature [4][5]. We will consider high speed ship (more than 40 knots) with 120 meters in length. Real data employed in this work have been obtained by the experimental canal CEHIPAR¹.

We have obtained an initial model approximation using a mechanic study of the ship. It is described by equation 1 and it establishes the relation between the system output (heave) and its input (sea waves) [6]:

$$G(s) = \frac{\text{Heave}(s)}{\text{Wave}(s)} = \frac{k \cdot (s^2 + s \cdot p_0 + p_1)(s^2 + s \cdot p_2 + p_3)}{(s + p_4) \cdot (s^2 + s \cdot p_5 + p_6) \cdot (s^2 + s \cdot p_7 + p_8)} \quad (1)$$

The problem consist of finding out the nine parameters p_i .

3 Parallel Genetic Algorithms

3.1 Outline

Genetic algorithms (GA) [7] are optimization techniques which imitate the way that nature selects the best individuals (the best adaptation to the environment) to create descendants which are more highly adapted. The first step is to generate a random initial population, where each individual is represented by a character chain like a chromosome and with the greatest diversity, so that this population has the widest range of characteristics. Then, each individual is evaluated using a fitness function, which indicates the quality of each individual. Finally, the best-adapted individuals are selected to generate a new population, whose average will be nearer to the desired solution. This new population is obtained applying three different operators: reproduction, crossover and mutation.

One of the major aspects of GA is their ability to be parallelised. Indeed, because natural evolution deals with an entire population and not only with particular individuals, it is a remarkably highly parallel process [8].

It has been established that GA efficiency to find optimal solution is largely determined by the population size. With a larger population size, the genetic diversity

¹ CEHIPAR: Canal de Experiencias Hidrodinamicas del Pardo, Madrid (Spain).

increases, and so the algorithm is more likely to find a global optimum. A large population requires more memory to be stored, it has also been proved that it takes a longer time to converge. The use of today's new parallel computers not only provides more storage space but also allows the use of several processors to produce and evaluate more solutions in a shorter time.

We use a coarse grained parallel GA. The population is divided into a few sub-populations or demes, and each of these relatively large demes evolves separately on different processors. Exchange between sub-populations is possible via a migration operator. In the literature, this model is sometimes also referred as the island Model. Sometimes, we can also find the term 'distributed' GA, since they are usually implemented on distributed memory machines. The code has been developed in C using the SGI implementation of MPI.

Technically there are three important features in the coarse grained PGA: the topology that defines connections between sub-populations, migration rate that controls how many individuals migrate, and migration intervals that affect how often the migration occurs.

Many topologies can be defined to connect the demes. We present result using a simple stepping stone model where the demes are distributed in a ring and migration is restricted to neighboring ones. We have used other models such as a master-slave one where a master population is connected to all the slaves but as we have obtained for other PGA algorithms, results are slightly worse in this case [9]. Moreover, ring topologies have a lower communication cost since the fewer the number of neighbours, the fewer the number of messages to be sent. This fact is more important in a workstation or PC cluster based on Ethernet where the network becomes a bottleneck.

Choosing the right time for migration and which individuals should migrate appears to be more complicated and a lot of work is being done on this subject. Several authors propose that migrations should occur after a time long enough to allow the development of goods characteristics in each sub-population[10]. However, it also appears that immigration is a trigger for evolutionary changes.

In our algorithm, migration occurs after a fixed number of generations that can be specified at execution time. They are selected from the best individuals in the population and they replace the worst ones in the receiving deme. The number of migrants can be selected at execution time as well.

3.2 Genetic Representation

As we explain on section 2, our algorithm goal is to identify the parameters of equation 1. Common techniques try to directly look for the values of the p_i parameters. One of the important features of genetic algorithms is that they allow to incorporate the knowledge of the problem to the code. We take advantage of that in our codification. Instead of looking for the parameters of the equations we will look for their roots, in order to reduce the searching space.

For example, in the denominator of equation 1, $(s^2 + p_5 s + p_6)$, p_5 and p_6 can take a value in $[-\infty, +\infty]$, but their roots must belong to $[-\infty, 0]$ because the system is desired to be stable. So we have eliminated a half of the searching space. In addition, as, the module of the roots must be of the same order than the maximum frequency of

encounter of the input waves (3 rad/s). So we will look for in $[-4,0]$. With that, we have delimited a reasonable searching space. The same occurs with the roots of the numerator, with the only difference that they can be positive.

Summing up, we have to find out the roots of the following functions:

$$\begin{aligned} (s^2 + p_0 s + p_1)(s^2 + p_2 s + p_3) & \quad \text{in } [-4,4] \\ (s + p_4), (s^2 + p_5 s + p_6), \text{ and } (s^2 + p_7 s + p_8) & \quad \text{in } [-4,0] \end{aligned} \quad (2)$$

The solution of these equations are complex numbers. We will denote them as z_1+z_2j , z_3+z_4j , d_1+d_2j , d_3+d_4j , and d_5+d_6j respectively.

An individual will be represented by a chromosome with 10 genes. A pair of genes represents the value of a root in the following order $[z_1 \ z_2 \ z_3 \ z_4 \ d_1 \ d_2 \ d_3 \ d_4 \ d_5 \ d_6]$.

We have chosen an integer codification for improving implementation performance. Thus, for a precision equal to 0.001, the alphabet is $\Omega = [-4000,4000]$, where 4000 represents 4.000, 3999 represents 3.999 and so on.

For instance, if we have a solution represented by the next individual:

$$[243 \ -4325 \ 1019 \ -2681 \ 2386 \ 0 \ -2912 \ -4283 \ -601]$$

the roots of functions (2) are $2.43 \pm 1.325j$, $1.019 \pm 2.681j$, $2.386 \pm 0j$, $-2.419 \pm 2962j$, and $1.283 \pm 601j$ and the model represented by the chromosome is:

$$G(s) = \frac{(s^2 - 0.486 \cdot s + 1.8147)(s^2 - 2.038 \cdot s + 8.2261)}{(s + 2.386)(s^2 + 4.838 \cdot s + 14.3313)(s^2 + 2.565 \cdot s + 2.0073)} \quad (3)$$

3.3 Fitness Function

We denote by EAm_i and EPh_i the experimental values of the amplitude and the phase of the ship heave for a set of frequencies (w_i) of the input wave, and by TAm_i and TPh_i the respective theoretical values. These theoretical values, which are obtained from each individual, must be as close to the experimental figures as possible, so our fitness function is:

$$J = \left(\sum_{\forall \omega_i} |EAm_i - TAm_i| + \sum_{\forall \omega_i} |EPh_i - TPh_i| \right)^{-1} \quad (4)$$

Thus, the algorithm will look for solutions that minimizes the difference between the experimental and the theoretical heave.

3.4 Selection Operator

The selection operator tackle the task of selecting the individuals of a generation for the crossover procedure. We have implemented one of the most widely used techniques; the wheel method [11]. So each individual will have a selection probability gives by equation 3.

$$\text{Selection_probabbility_of_individual_i} = \frac{J_i}{\sum_i J_i} \quad (5)$$

Where J_i is the value of the fitness function computed for individual i .

3.5 Crossover and Mutation Operators

The crossover operator mix the features of 2 individuals by the combination of their genes. There are several crossover operator described in the literature, but we have implemented a simple one-point crossover operator, because we have obtained good results in other optimization problems with the same operator and it has the easiest implementation.

In the other hand the mutation operator allows genetic algorithms to avoid local optimums. It makes random changes with a very low probability. In this way we can explore additional areas of the searching space.

3.6 Performance Evaluation of the Parallel Genetic Algorithm

The purpose of this evaluation is to measure the speed-up when running the PGA. We use the speed-up ratio as a metric for the performance of the parallel genetic algorithm. But we use an special speed-up metric instead of the classical time speed-up. We define the speed-up S in terms of the number of generations:

$$S = \frac{N_s}{N_p} \quad (6)$$

Where N_s is the number of generations used by the algorithm to reach a fixed value on a single processor and N_p is the number of generations for a p processors implementation. We use the number of generations because it gives us a measure not only of the time speed-up, but also of the algorithm convergence properties.

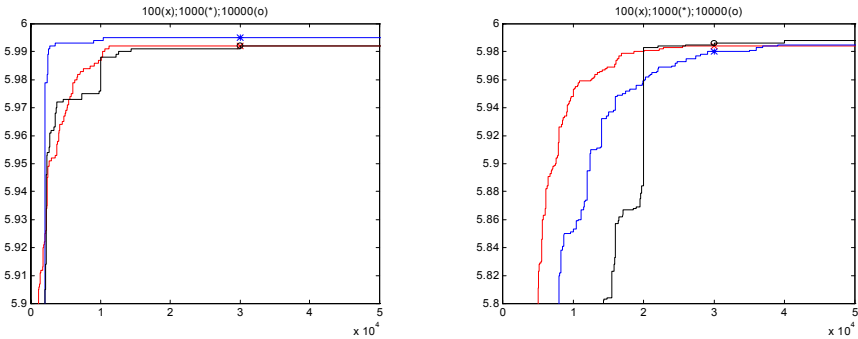


Fig 1. Epoch length study. The figures represent the fitness function with the number of generations for 8 processors (on the left chart) and 4 processors (on the right chart) respectively.

3.7 Epoch Lengths

As we have explained before, there are several features which influence the algorithm efficiency and convergence. One of the most important is the epoch length. If the interchange between demes is frequently accomplished the genetic algorithm could loose their efficiency and we could make a quasi-random search. In the other hand if we made few interchanges we will not take advantage of the parallel genetic algorithm and it evolves as a set of sequential and independent algorithms. So we must reach a trade-off between epoch length and the number of processors [12].

Figure 1 compares the fitness function with the number of processors, for 100, 1000 and 10000 epoch lengths for eight (on the left chart) and four (on the right chart) processors respectively. We can see that for the 8 processors implementation we need to interchange migrants more frequently if we want to exploit the PGA features since the algorithm uses a population with more individuals. In the other hand for a 4 processors implementation the epoch length must have a medium value.

4 Performance Results

Figure 2 shows the PGA speed-up using a local population size of 20 individual, i.e. 20 individual on each processor. Initial solution has been obtained randomly using a different seed on each processor and migration occurs every 100 generations.

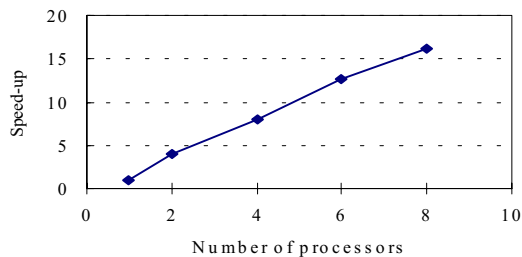


Fig 2. Speed-up of the Parallel genetic algorithm in terms of the number of generations

For the 20 individuals algorithm, the PGA achieves high speed-up. However, increasing the local population improves the sequential algorithm results and thus, deteriorates the Speed-up. However the generation cost also increases.

5 Experimental Results

After the application of the PGA we obtained the next model description:

$$G(s) = \frac{\text{Heave}(s)}{\text{Wave}(s)} = \frac{(s^2 - 1.04 \cdot s + 4.2864)(s^2 + 0.206 \cdot s + 9.7710)}{(s + 3.978) \cdot (s^2 + 2.318 \cdot s + 7.742) \cdot (s^2 + 0.656 \cdot s + 8.8810)} \quad (7)$$

Figure 3 and 4 show the results obtained for the heave with the PGA model. The first one corresponds to an irregular wave and the second one compares the model with a regular wave, its correspondent experimental heave and the theoretical heave obtained by the PGA. As figure shows, the theoretical results reproduce accurately real data.

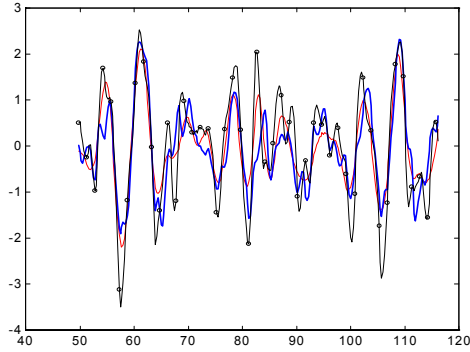


Fig. 3. Comparison between the heave modeled with the PGA and the experimental data for an irregular input wave.

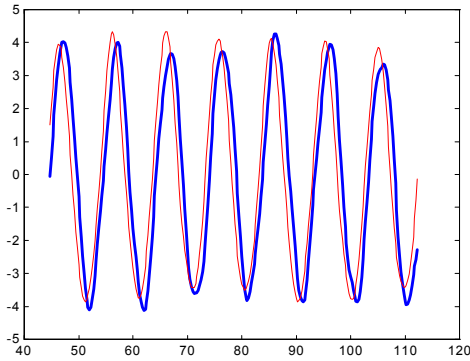


Fig. 4. Comparison between the heave modeled with the PGA and the experimental data for a regular input wave.

6 Conclusions and Future Work

The main conclusions can be summarized as follows:

- We have designed a new method for identifying model parameters using PGA.
- Our method is perfectly adaptable to other models. We only need to know the parameter range.
- The PGA performance depends on the PGA parameters choice. A correct election allows us to obtain high parallel efficiencies and so, assure time constrains.

As future work we propose to compare the coarse grain parallelization with the parallelization of the fitness function (the most expensive task of our algorithm) and the application of this method to other problems.

Acknowledgments

This work has been supported by the Spanish research grants TAP 97-0607-C03-01 and TIC 96-1071.

We would like to thank Centro de Supercomputacion Complutense for providing access to the parallel computer used in this research.

References

1. Van den Bosch, P.P.J., van der Klauw, A.C. : Modeling, Identification and Simulation of Dynamical Systems. CRC Press, London (1994).
2. Cantú-Paz, E.: A Survey of Parallel Genetic Algorithm. IlliGAL Report No. 97003. University of Illinois at Urbana-Champaign (1997).
3. Fossen, T. I.: Guidance and Control of Ocean Vehicles. J.Wiley and Sons. New York (1994)
4. Lewis, E.V. : Principles of Naval Architecture. The Soc. of Naval Architects and Marine Engineers. New Jersey (1989).
5. Lloyd, A.R.J.M. : Seakeeping: Ship Behavior in Rough Weather. Ellis Horwood (John Wiley and Sons). New York (1988).
6. De la Cruz, J.M., Aranda, J., Ruiperez, P., Diaz, J.M., Maron, A.: Identification of the Vertical Plane Motion Model of a High Speed Craft By Model Testing in Irregular Waves. IFAC Conference CAMS'98 Fukuoka (1998).
7. Davis: Handbook of Genetic Algorithms Van Nostrand Reinhold (1991).
8. Bertoni, A. Dorigo, M. : Implicit Parallelism in Genetic Algorithms. Artificial Intelligence (61) 2, (1993) 307-314 .
9. Hidalgo, J.I., Prieto, M., Lanchares, J., Tirado, F.: A Parallel Genetic Algorithm for solving the Partitioning Problem in Multi-FPGA systems. Proc. of 3rd Int. Meeting on vector and parallel processing. Porto (1998) 717-722.
10. Liening, J. : A Parallel Genetic Algorithm for Performance-Driven VLSI Routing. IEEE Trans on Evolutionary Computation VOL.1 NO.1 April 1997.
11. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs. 3rd edn. Springer-Verlag, Berlin Heidelberg New York (1996).
12. Hüb, X. : Genetic Algorithms for Optimisation Background and Applications. Edinburgh Parallel Computing Center. Version 1.0 February 1997. Available from: <http://www.epcc.ed.ac.uk/epcc-tec/documents/> .

Large-scale FE modelling in geomechanics: a case study in parallelization

Radim Blaheta¹, Ondřej Jakl^{1,2}, Jiří Starý²

¹ Institute of Geonics of the Czech Acad. Sci.,
Studentská 1768, 708 00 Ostrava – Poruba, Czech Republic
{[blaheta](mailto:blaheta@ugn.cas.cz),[jakl](mailto:jakl@ugn.cas.cz),[stary](mailto:stary@ugn.cas.cz)}@ugn.cas.cz

² VŠB – Technical University Ostrava,
ul. 17. listopadu, 708 00 Ostrava – Poruba, Czech Republic

Abstract. The paper develops the Displacement Decomposition technique used already in [1] for parallelization of the Preconditioned Conjugate Gradient method. This time, the solution of very large problems is tackled. We present our experience with (PVM-)parallel solution of a huge finite element model arising from geomechanical practice on usual hardware and describe various facets of the optimizing process, leading to a tenfold improvement of the solution time. Special effects of parallelizing I/O-constrained programs are also addressed.

1 Introduction

In [1] we presented our first results in the PVM-parallelization of the PCG solver of an in-house finite element package *GEM32*, and indicated that the parallel version might be especially useful to cope with large tasks. Since then, we embarked upon the solution of a complex geomechanical problem related to mining in the uranium deposit at Dolní Rožínka (DR), which required repeated solution of a large model of almost four million degrees of freedom. This task motivated further developments of our FE code and its parallel solver in particular.

2 The PCG solver and its parallelization

The original solver *PCG-S* of the *GEM32* package and its successors implement the *Preconditioned Conjugate Gradient (PCG) method* for the solution of linear systems $Au = f$, where A is symmetric, positive definite $N \times N$ *stiffness matrix*, u, f are N -dimensional vectors. The Conjugate Gradient algorithm can be easily partitioned by decomposing the matrix A into blocks, but the preconditioning transformation has to be treated with care.

2.1 Preconditioning

In the PCG algorithm, *preconditioning* improves the search directions for the solution. The characteristic of the PCG-S solver is the so-called *Displacement Decomposition – Incomplete Factorization (DiD-IF)* preconditioner. As explained

in [1], the DiD is based on separation of the three components of the nodal displacement vector and has favourable properties for the parallelization. Namely, it provides a completely decomposed preconditioner C with three diagonal blocks C_k ($k = 1, 2, 3$) having properties that enable their incomplete factorization and imply easy solution of the $C_k w_k = r_k$ preconditioning operation. This allows a direct coarse-grained partitioning to three concurrent tasks.

An alternative to the incomplete factorization is a low accuracy solution of the $C_k w_k = r_k$ systems by an another, inner PCG iteration processes. We call this modification *DiD-IE* preconditioning. The main advantage for the parallel solution is expected in the reduction of the expensive outer iterations resulting from the migration of the computational work to communication-free inner iterations. Such decrease of the communication/computation ratio should be valuable especially on slow networks. The benefits of DiD-IE for the sequential solution are currently under investigation. For more mathematical issues see [2].

2.2 Parallelization

The layout of the iterative phase of the parallel PCG algorithm based on the Displacement Decomposition is given in Fig. 1. This algorithm gives rise to three *workers*, each of which processes data corresponding to one displacement direction, and a computationally inexpensive *master*. The so called “*big*” *communication* is necessary in each iteration for the matrix–vector multiplication. Here, six vectors of $4N/3$ Bytes must be exchanged between the workers. Other data transfers are negligible.

The parallel realization *PCG-P* implements both the DiD-IF and the DiD-IE preconditioning. For short, we refer to those two run-time alternatives as *PCG-P(IF)* and *PCG-P(IE)*. PCG-P is coded independently of a particular message passing system, and has interfaces to both the PVM and MPI libraries.

2.3 Disk I/O

To maximize the size of solvable problems, the stiffness matrix is not processed in memory, but repeatedly read row/record-wise during operations like matrix–vector multiplication, relying on file caching of modern operating systems. Although regular grids of GEM32 reduce the storage requirements to mere 42 non-zero elements per row, its size e.g. for the DR model is about 650 MB. This implies that the code is I/O-bound. Moreover, a considerable amount of storage is allocated for necessary data structures. The solver may soon exhaust available RAM and force the operation system to use virtual memory. For large practical problems, this is the rule on our hardware, resulting in 80–90 % waiting time of the CPU.

In this context, splitting the computation among several computers with local disk storage is supposed to yield more benefit than just a speedup approaching the number of processors. It brings concurrence in the time-consuming I/O operations and with smaller subtasks, reduces or even eliminates paging due to insufficiency of real memory.

Table 1. Solution times for the Neumann modelling sequence, in hours:minutes

Solver\Step	Step 1	Step 2	Step 3	Step 4	Total time
PCG-S	25:54	05:40	04:28	09:14	45:16
PCG-P(IF)	11:55	02:42	02:36	04:46	21:20

model tried to take into account the pre-mining stress state, anisotropic according to measurements. This lead to four pure Neumann boundary value problems, where the boundary conditions simulate the action of the surrounding rock. We denote this four-step task *Neumann Modelling Sequence (NMS)*.

Table 1 shows the times obtained in our first experiments. The relative accuracy was set to 10^{-3} . In steps 2, 3 and 4, the result of the previous step was taken as an initial guess. Although the parallel code shortened the execution (relative speedup of 2.1), more than 45 and 21 hours respectively (plus 3 and 4.5 hours overhead) for PCG-S and PCG-P(IF) are hardly acceptable for routine recomputations. Moreover, the PCG-P(IE) solver diverged for NMS.

3.2 Computing resources

If not stated otherwise, all the computations referenced in this study were performed on an entry-level installation of the well-known IBM SP multicomputer, equipped with eight POWER2/66.7 MHz processor nodes with 128 MB of system memory. Each node had three network interfaces, Ethernet 10 Mbit/sec, ATM 155 MBit/sec and the proprietary High Performance Switch (HPS) 40 MByte/sec. Software environment included AIX 4 and Parallel Virtual Machine (PVM) version 3.3.11.

The measurements were carried out in non-dedicated mode, but in periods without other (non-system) applications, in case of doubts repeatedly. The wall-clock times are reported. The PVM version (used in the RS6K mode) of the PCG-P solver was employed. For technical problems with HPS in the period of DR solution, the ATM interconnect was used instead.

4 Improving the PCG performance

When optimizing the performance of a parallel system, i.e. the combination of a parallel program and its target machine, one has to consider all aspects known from sequential programming, but also the communication issues. These include:

- the theory behind the algorithm, to reduce the communication as such,
- the implementation of the algorithm in the given message passing system, to realize the message passing efficiently in its environment,
- the execution of the program, to take most of the communication capabilities of the underlying hardware and operating system.

Having primarily the DR problem in mind, we tried to improve the PCG performance by investigating diverse items of the above scope.

4.1 The Dirichlet modelling sequence

The Neumann boundary problems are not very advantageous for iterative solvers. Moreover, we had to cope with *singular systems*, which additionally can be (*slightly*) *inconsistent*. Therefore, the generalized solution has to be sought. In the case of slightly inconsistent systems, the initially converging PCG method starts to diverge and needs to be stabilized by *projections*.¹

Thus, the first step towards optimization of the DR computation was seen in eliminating the Neumann steps from the modelling sequence. Further research suggested the possibility of replacing NMS by the *Dirichlet modelling sequence* (DMS): It starts with an auxiliary Neumann problem to get displacements which are then applied as boundary conditions in the following four pure Dirichlet boundary problems, for which the iterative solution is as a rule very efficient and robust. See Table 2. From the comparison with Table 1 follows that both

Table 2. The Dirichlet modelling sequence, in hours:minutes

Solver\Step	Step 0	Step 1	Step 2	Step 3	Step 4	Total time
PCG-S	25:45	01:03	00:36	00:36	01:02	28:17
PCG-P(IF)	12:00	00:29	00:16	00:15	00:30	13:30

the PCG-S and PCG-P(IF) solvers needed only about 60% of the NMS time to complete DMS, maintaining a speedup of about 2.1 of the parallel version.

4.2 DiD-IE preconditioning

A decisive progress in the speed of the solution of the DR problem was achieved by applying the DiD-IE preconditioner. To be able to cope with singular systems like that one in Step 0 of DMS, the PCG-P solver had to be enriched with projection option, which stabilizes the iterative process. The results are shown in Table 3. We are looking for the explanation of this enormous speedup (4.3 and 9 respectively regarding PCG-P(IF) and PCG-S in Table 2) in (1) communication reduced by order of magnitude and (2) less disk I/O due to greater data locality: The computation work concentrates in the solution of smaller subproblems corresponding to the diagonal blocks of the stiffness matrix (one ninth of

Table 3. The Dirichlet modelling sequence, DiD-IE preconditioning. Speedup of 9!

Solver\Step	Step 0	Step 1	Step 2	Step 3	Step 4	Total time
PCG-P(IE)	02:12	00:19	00:12	00:12	00:14	03:09

¹ The exact mathematical treatment can be found in [2].

its memory demands). See also Sect. 4.4. Anyway, thanks to parallelization, the solution of the DR sequence, being 14.6 times shorter than at the beginning, is now much more acceptable for practical engineering analyses.

4.3 Message passing optimizations

A good parallel algorithm does not automatically mean a good parallel program — in fact, because of the extend of parallel libraries, the process of coding and optimization is usually a fine job full of experiments. In message-passing applications, the crucial point is the design and realization of efficient communication patterns.

In this respect, the “big” communication (see Sect. 2.2) is critical in PCG-P. Fig. 2 shows two variants of its realization under PVM. In the original solution (upper diagram), the vector blocks were distributed through the master, whereas at present (lower diagram), non-blocking receives enable concurrent data transfers directly between the pairs of workers. The effect of this change is about 50% reduction of the communication time, most beneficial with slow networks (Ethernet) and iteration-rich computations (PCG-P(IF)).

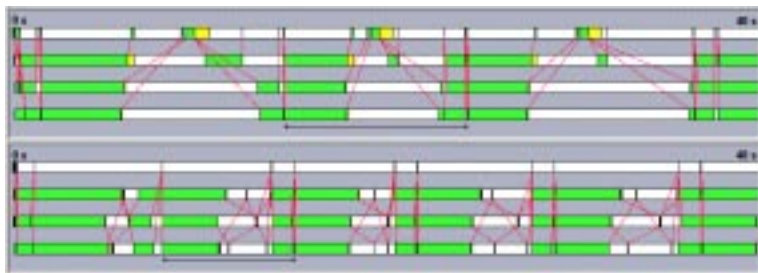


Fig. 2. Snapshots of two XPVM Space-Time diagrams demonstrate the computation flow of the PCG-P(IF) solver and improvement of the “big” communication. First rows belong to the master process. Second iterations are delimited (*below*)

4.4 Comparing network interconnections

The IBM SP system, having three network interfaces, allowed us to make some additional experiments showing the influence of the underlying communication network on the parallel solver’s performance. This is of practical importance since we want the code to be usable e.g. on an Ethernet cluster of workstations.

The values in Table 4 were obtained when timing the last step in DMS. To reduce the influence of the initialization phase of the iterative process, no initial guess was applied and the relative accuracy was changed to 10^{-4} . Note that there is almost no difference in times measured on the particular networks.

Table 4. Averaged wall-clock times (in seconds) for the modified fourth step of DMS. Three interconnection networks tested, with experimentally (in PVM, IP protocol) measured bandwidth (*second column*)

Network type	Bandwidth B	PCG-P(IF)	PCG-P(IE)
	MByte/sec	$I = 71$ iter.	$I = 7$ iter.
Ethernet	1.0	18 073	3 293
ATM	7.5	17 330	3 303
HPS	9.0	17 303	3 236

The point is that the communication/computation ratio is very low when disk operations dominate. According to a rough model, the total communication time is expected to be about $(8NI)/(3 \cdot 10^{20}B)$ seconds (N is the size of the linear system, I number of iterations, B bandwidth in MByte/sec), which makes for PCG-P(IF) and PCG-P(IE) respectively circa 700 and 70 sec on the Ethernet. This is much less then the variations in the execution time, with large tasks observed as much as 15%. (In fact, larger bandwidth is taken advantage of in data distribution phase, preceding the parallel computation.)

We verified this qualitative fact on a sequence of FOOT benchmarks (introduced in [1]) which we scaled up to the size of the DR problem. From the resulting graphs in Fig. 3, 4 one can derive the following observations:

1. The PCG-P(IE) solver outperforms the PCG-P(IF) solver on both the Ethernet and HPS networks, increasing its advantage with the size of the FOOT problem almost to a factor of 4, with the fourth DR step even to 5.3. The slower is the communication, the better are the times of PCG-P(IE) relatively to PCG-P(IF) also for small problems.
2. With both interconnections, the execution times converge with the increasing size of linear systems, but for small tasks (no disk activity) are in the proportion of 2 : 1 for PCG-P(IF) and of 4 : 3 for PCG-P(IE). For very large tasks and PCG-P(IF), we encountered sometimes even slightly better performance on Ethernet than on HPS (dispersion of long measurements?).
3. In our computing environment (Sect. 3.2), the critical size of the linear system is about one million unknowns, when the parallel solvers begin to be retarded by disk operations.

5 Conclusions

This paper reported upon the history of the solution of a large FE model, when the parallelization method, theoretically limited to a speedup of three, resulted in a solution time fifteen times shorter than the starting one. No miracles — mathematical reformulations contributed significantly to this improvement, but at the same time, one less reported beneficial aspect of parallelization has been clearly exposed: Namely better hardware utilization due to partitioning to sub-problems more appropriate in size, leading to a superlinear speedup. Although

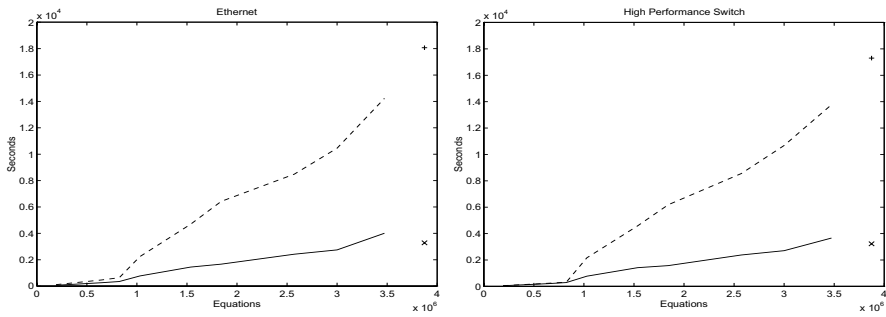


Fig. 3. Interpolated solution times for nine FOOT tasks (192 000 – 3 472 875 equations), solvers PCG-P(IF) (*dashed*) and PCG-P(IE) (*solid*). Ethernet (*left*) and HPS (*right*). DR times (*plus and x-mark*) as reference

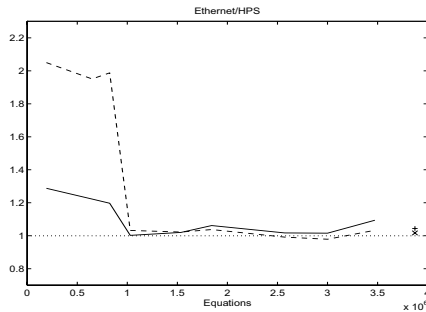


Fig. 4. The Ethernet/HPS solution time ratio for the PCG-P(IF) (*dashed*) and PCG-P(IE) (*solid*) on the FOOT sequence. DR ratio (*plus and x-mark*) for illustration

perhaps not so interesting from the theoretical point of view, it is extraordinary important for practical computations like the reported one.

Acknowledgement

Supporting contracts: EC COPERNICUS 977006, Czech Ministry of Education No. LB98212, LB98273 and VŠB – TU Ostrava CEZ:J17/98:2724019.

References

1. Blaheta, R., Jakl, O., Stary, J.: PVM-implementation of the PCG method with Displacement Decomposition. In: Bubak, M, Dongarra, J., Wasniewski, J. (eds.): Recent Advances in Parallel Virtual Machine and Message Passing Interface. Lecture Notes in Computer Science, Vol. 1332, Springer-Verlag, Berlin, (1997) 321–328
2. Blaheta, R. et al.: Iterative displacement decomposition solvers for HPC in geomechanics. Submitted to Large-Scale Scientific Computations, Sozopol, Bulgaria

A Parallel Robust Multigrid Algorithm Based on Semi-coarsening

M. Prieto, R. Santiago, I.M. Llorente, and F. Tirado

Departamento de Arquitectura de Computadores y Automatica
Facultad de Ciencias Fisicas, Universidad Complutense, 28040 Madrid, Spain
{mpmatias,rubensm,lllorente,ptirado}@dacya.ucm.es

Abstract. This paper compares two common approaches to achieve robustness in the parallel multigrid resolution of anisotropic operators on structured grids: alternating plane smoothers with full coarsening and plane smoothers combined with semicoarsening. Both, numerical and architectural properties are compared. A parallel implementation based on MPI is studied in a realistic way by considering the exploitation of the cache memory and the negative effect that the parallelization has on the convergence properties of the methods.

1 Introduction

Several methods have been proposed in multigrid literature to solve anisotropic operators and achieve robustness when the coefficients of a discrete operator can vary throughout the computational domain (due to grid stretching or variable coefficients). See [5] for a comparison between different methods.

Single-block grids are widely used in the numerical simulation of many Physical systems. Single-block algorithms are very efficient (especially for structured grids with stretching) due to their relatively easy implementation (on both sequential and parallel computers) and good architectural properties (parallelism and cache memory exploitation). Moreover, single-block grids are used as building blocks for multi-block grids that are needed to deal with complex geometries or to facilitate parallel processing.

In this paper we compare two common approaches to achieve robustness on single-block structured grids: alternating-plane smoothers combined with standard coarsening [1,5] and plane smoothers with semi-coarsening [2] algorithms. We study the numerical and architectural properties of both approaches for the solution of a 3D discrete anisotropic elliptic model problem on stretched cell-centered grids.

This paper is organized as follows. The numerical scheme used by our simulations is described in section 2. The numerical results (convergence factor) obtained for different anisotropic problems are found in section 3. Architectural properties (memory hierarchy use and parallel efficiency) are presented in sections 4 and 5 respectively. The paper ends with some conclusions.

2 The Numerical Problem

As a test problem, we have consider the multigrid solution of the volume finite discretization of the diffusion model equation. Multigrid [8] is based on two fundamental principles, relaxation and coarse grid correction. The relaxation procedure is based on a standard iterative method, called a smoother in multigrid literature. It is used to damp high frequency or oscillatory components of the error, since it fails in attenuating low-frequency components. To solve this, a coarser grid correction is used, because on a suitably coarser grid, low-frequency components appear more oscillatory, and so the smoother can be applied effectively.

Regular multigrid (based on point-wise smoothers) is not suitable for a general anisotropic problem. In order to solve such a problem, we have to change either the relaxation method or the coarsening procedure. Several ways have been proposed in literature to make multigrid a robust solver. Now, we will focus on the two alternatives under study in this paper:

Alternating Plane Smoothers: A general rule in multigrid is to solve simultaneously those variables which are strongly coupled by means of plane or line relaxation [1]. In a general situation the nature of the anisotropy is not known beforehand, so there is no way to know which variables are coupled. One solution consists in exploring all the possibilities, i.e. to use plane relaxation in the three possible coordinate directions.

Semi-coarsening: Several robust methods have been proposed based on changing the coarsening strategy. Among them, we have considered the multigrid scheme presented in [2]. For a 2D problem, the basic idea is simply to use x-line relaxation and y-semicoarsening (doubling the mesh size only in the y-direction) together, or in the same way, to use both y-line relaxation and x-semicoarsening. This approach extends naturally to 3-D in the form of xy-plane relaxation (xz-plane, yz-plane) and z-semicoarsening (y-semicoarsening, x-semicoarsening).

The test code has been developed in C. The system of equations is solved by the full approximation scheme (FAS). This multigrid method is more involved than the simpler correction scheme but can be applied to solve non-linear equations. Both approaches (alternating plane and semi-coarsening) have been implemented in an "all-multigrid" way. The block solver (2D-solver) is a robust 2D multigrid algorithm based on full coarsening and alternating line smoothers. The Thomas' algorithm or 1D Multigrid is used to solve the lines.

Lexicographic Gauss-Seidel relaxation is always used in the following experiments. Test cases have been performed on a 64x64x64 grid and a general (1,1) V-cycle is used whenever multigrid is applied. To reduce roundoff errors, we have fixed the right-hand side of the equation and the appropriate Dirichlet boundary conditions such that its solution is zero.

3 Convergence Rate

In order to compare the robustness of both approaches we consider a 3D Cartesian grid with exponential stretching in the three dimensions. Robustness is absolutely necessary in this example since the anisotropy varies throughout the whole domain. Although anisotropy sources can be either physical (diffusion coefficients) or computational (grid stretching), the example is general since the effect on the system of equations is equivalent for both sources.

As figure 1 (left-hand chart) shows, both methods solve the problem effectively. For the semi-coarsening approach, each coarsening direction exhibits the same performance since a dominant direction does not exist. The alternating plane smoother improves its convergence factor as the anisotropy grows (as was shown in [5]).

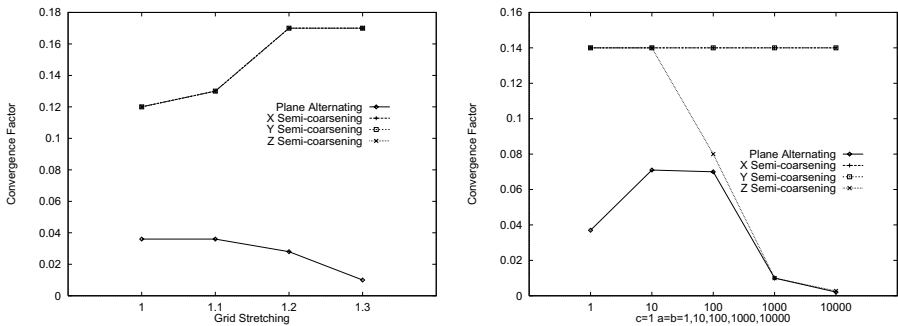


Fig. 1. Convergence factor for the isotropic equation and different grid stretching factors (left-hand chart), and anisotropic equation on a uniform grid ($au_{xx} + bu_{yy} + cu_{zz} = 0$) for several coefficient sets (right-hand chart)

On the other hand, when the anisotropy is located on one plane the best coarsening procedure is the one that maintains coupling of strongly connected unknowns. For example if we make the coefficients grow in the xy-plane (see figure 1, right-hand chart), z-coarsening gets a better convergence factor because the smoother becomes an exact solver for high anisotropies. Alternating plane smoothers presents a similar behavior, it becomes a direct solver when the unknowns are strongly coupled, i.e. the better the plane is solved the better obtained convergence.

In conclusion, we can state that the alternating plane smoother approach exhibits a better convergence factor than the semi-coarsening one. However, we should note that the operation count per multigrid cycle is higher and so the convergence per working unit must be used as a definitive metric to compare both approaches.

4 Memory Hierarchy Exploitation

When we are trying to compare two different algorithms, one has to take into account not only their numerical efficiency, but also their architectural properties. It is well known that the better a program exploits the spatial and temporal locality the less time it takes. Indeed, the maximum performance that can be obtained in current microprocessors is limited by the memory access time.

Using the SGI perfix profiling tool, we have measured that the number of L1 and L2 cache misses of the alternating plane smoother are in general greater than the semi-coarsening. Although not all misses incur the same timing penalties in current microprocessors, they can be used for evaluating how the algorithms exploit the memory hierarchy. The measurements have been taken on two different systems with the same R10000 processor running at 250 MHz, an O2 Workstation and one processor of an Origin 2000 system (O2K). This processor has a 32 KB primary data cache, but the external L2 cache size varies: 1MB on the O2 and 4MB on the O2K. We should also note that the code has been compiled using different optimization options on both systems: Ofast=ip32_10k on the O2 and Ofast=ip27 on the O2K.

For a 32x32x32 problem size, the number of L1 misses (around the same on both systems) of x-semicoarsening approach are about 12% and 50% fewer than those obtained for the y-semicoarsening and the alternating approach, respectively. These differences grow slightly for the 64x64x64 problem size.

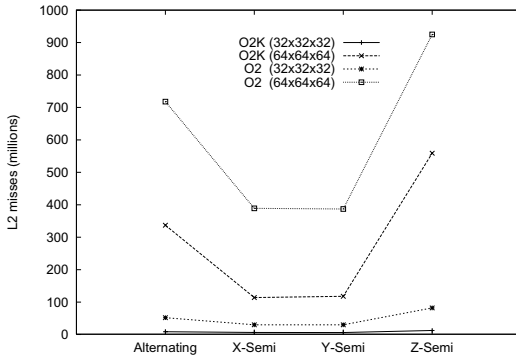


Fig. 2. Millions of L2 cache misses for the 64x64x64 and 32x32x32 problem sizes on an O2000 system and O2 workstation

Figure 2 shows the number of L2 cache misses on both systems for 32x32x32 and 64x64x64 problem sizes. It is interesting to note that the alternating approach and the z-semicoarsening have around 1.75 and 2.5 times more misses than the x-semicoarsening on the O2 system for both problem sizes. However, on the O2K system, where the large second level cache allows a better exploitation of the temporal locality, the differences grow with the problem size since,

for smaller problems, the spatial locality has less impact on the number of cache misses. The greatest differences between both systems are obtained for the x and y-semicoarsening (around 3.4 times more misses on the O2 system) due to temporal locality effects, which are lower on the alternating an z-semicoarsening approaches where the spatial locality has more influence.

Therefore, in order to make a more realistic comparison between the two methods and the different semi-coarsening directions, we have to measure the convergence factor per working units (WU) (the WU has been defined as the execution time needed for evaluating the equation metrics on the finest grid). As figure 3 shows the alternating plane smoother has a better behavior than the semi-coarsening, i.e. it reduces the same error in less time.

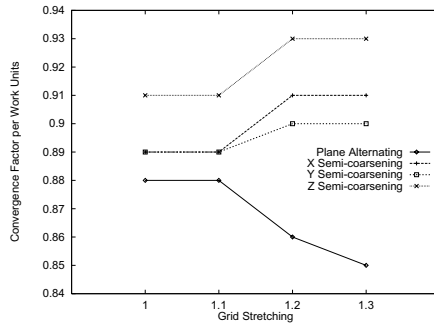


Fig. 3. Convergence factor per Working units (WU) in the 64x64x64 stretched grid.

5 Parallel Properties

5.1 Outline

To get a parallel implementation of a multigrid method, one can adopt one of the following strategies (see, e.g., [6]).

1. Domain decomposition: domain decomposition is applied first. Then, a multigrid method is used to solve problems inside each block.
2. Grid partitioning: domain decomposition is applied on each level.

Domain decomposition methods are easier to implement and imply fewer communications since they are only required on the finest grid. However, the decomposition methods lead to algorithms which are numerically different from the sequential version and they have a negative impact on the convergence rate. Grid partitioning retains the convergence rate of the sequential algorithm. However it implies more communication overheads since exchange of data are required on each grid level.

Alternating-plane presents worse parallel properties than the other robust approach analyzed since, regardless of the data partitioning applied, it requires the solution of tridiagonal systems of equations distributed among the processors. The parallel solution of this kind of systems has been widely studied, see for example [4]. A specific solution for 2D alternating-line problems based on a pipeline Thomas's algorithm that could be extended for our 3D problem has been analyzed on [3]. The parallel efficiency of this algorithm obtained on a Cray T3E using MPI for problem sizes range from 256×256 to 1024×1024 cells varies between 0.6 to 0.8 respectively. However, current memory limits do not allow us to solve 3D problems where their corresponding 2D planes are big enough to obtain satisfactory efficiencies on medium-sized parallel computers.

Since the semi-coarsening approach does not need the alternating plane smoothers, an appropriate 1D grid partitioning and plane-smoother combination can be chosen so that a parallel tridiagonal solver is not needed.

5.2 Experimental Results

The parallel code has been developed using the Cray MPT (Message Passing Toolkit) version of MPI and the experimental results have been obtained on a Cray T3E-900 using up to 16 processors. We use -O3 optimization level for compiling the programs. Common grid transfer operators such as bilinear (2D) or trilinear (3D) interpolation and full weighted restrictors are parallel by nature. Hence, they do not suffer any change on the parallel code. However, the lexicographic Gauss-Seidel smoothers employed in the previous sections can no longer be applied, so they have been replaced by plane relaxation in a red-black zebra order of planes (slightly changing the numerical properties of our parallel code [5]).

Since the linear system of equations has to be solved "exactly" on the coarsest grid (by means of an iterative solver), it is usually chosen as coarse as possible to reduce the computational cost. However, from a parallel point of view, this decision may cause some processors to remain idle on the coarsest grids, reducing the parallel efficiency of the multigrid algorithm. Hence, how to avoid idle processors is an important issue in developing efficient parallel multigrid methods.

In our code, one approach known as the parallel U-Cycle method [9] has been used. The idea is simple, the number of grid levels has been fixed so that the coarsest grid employed on each processor has one plane. Figure 4 (left-hand chart) presents the parallel efficiency obtained on the T3E using five iterations of the smoother on the coarsest grid. The efficiency has been measured using the time spent on a multigrid cycle.

If a dominant direction does not exist, y-semicoarsening and x-semicoarsening are more efficient from an architectural point of view, since they exhibit a better spatial locality. For the same reasons, they are also the best choice from a message passing point of view, since message passing performance also depends on the spatial locality of the messages [7]. Using the Apprentice profiling tool, we have measured that the time spent performing data cache operations on a two processor simulation using z-partitioning, i.e. using z-semicoarsening, is about

17% larger than in the x-partitioning for a 64x64x64 problem size. This difference is lower than in the SGI systems analyzed above due to better behavior of the T3E memory hierarchy when spatial locality does not exist [7].

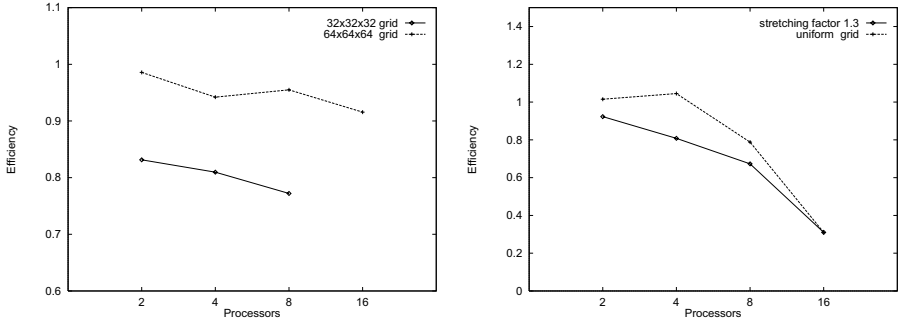


Fig. 4. Parallel efficiency obtained on a Cray T3E up to sixteen processors.

However, the number of iterations needed to solve the system of equations on the coarsest problem grows with its size and five iterations may not be enough for obtaining the exact solution. So, parallel efficiency has to be expressed using the execution time needed for solving the whole problem, i.e. reaching a certain residual norm (we have chosen 10^{-12}). Figure 4 (right-hand chart) shows the realistic parallel efficiency obtained on a Cray T3E

The parallel efficiency obtained is satisfactory using up to eight processors. In the sixteen processor simulation the efficiency decreases since the iterative method used for solving the equation on the coarsest level needs to converge a number of iterations that grow with the system size. As a future research, 2D and 3D decompositions will be used, since they allow to reduce the coarsest problem size. Due to convergence properties, isotropic simulation also have better parallel efficiencies than the anisotropic cases.

6 Conclusions and Future Research

Two common multigrid robust approaches for anisotropic operators have been compared taking into account both numerical and architectural properties:

- Both alternatives present similar convergence rates for low anisotropies. However, the alternating-plane smoothing process becomes an exact solver for high anisotropies and so, the convergence factor tends to zero with an increasing anisotropy.
- With regard to the cost-per-cycle, the alternating-plane approach is about a 38% more expensive. This difference depends not only on the number of

operations but also, on the memory hierarchy use. This fact is also important in the parallel setting because communication costs depend also on the spatial locality of the messages.

- The memory requirements of the semi-coarsening approach are about twice larger than the alternating-plane one.
- Parallel implementation of the alternating-plane approach involve the solution of distributed planes and lines that make them complicated. These difficulties can be avoided by the semi-coarsening approach and a proper linear decomposition. However, using an U-Cycle technique in order to avoid idle processors, the parallel efficiency obtained is not satisfactory due to the number of iterations required for solving the system on the coarsest grid. As a future research we will implement others possibilities such as the alternating-plane approach and a true V-Cycle since it allows to reduce the coarsest problem size.

Acknowledgments. This work has been supported by the Spanish research grants TIC 96-1071 and TIC IN96-0510 and the Human Mobility Network CHRX-CT94-0459. We would like to thank Ciemat and CSC (Centro de Supercomputacion Complutense) for providing access to the systems that have been used in this research.

References

1. A. BRANDT, *Multigrid techniques: 1984 guide with applications to fluid dynamics*, Tech. Rep. GMD-Studien 85, May 1984.
2. J. E. DENDY, S. F. MCCORMICK, J. RUGE, T. RUSSELL, AND S. SCHAFFER, *Multigrid methods for three-dimensional petroleum reservoir simulation*, in Tenth SPE Symposium on Reservoir Simulation, february 1989.
3. D. ESPADAS, M. PRIETO, I. M. LLORENTE, AND F. TIRADO, *Parallel resolution of alternating-line processes by means of pipelined techniques*, in Proceedings of the 7th. Euromicro Workshop on Parallel and Distributed Systems, IEEE Computer Society Press, Febraury 1999, pp. 289–296.
4. KRECHEL, PLUM, AND STÜBEN, *Parallelization and vectorization aspects of the solution of tridiagonal linear systems*, Parallel Comput., 14 (1990), pp. 31–49.
5. I. M. LLORENTE AND N. D. MELSON, *Robust multigrid smoothers for three dimensional elliptic equations with strong anisotropies*, Tech. Rep. 98-37, ICASE, 1998.
6. I. M. LLORENTE AND F. TIRADO, *Relationships between efficiency and execution time of full multigrid methods on parallel computers*, IEEE Trans. on Parallel and Distributed Systems, 8 (1997), pp. 562–573.
7. M. PRIETO, I. LLORENTE, AND F. TIRADO, *Partitioning regular domains on modern parallel computers*, in Proceedings of the 3th International Meeting on Vector and Parallel Processing, J. M. L. M. Palma, J. Dongarra, and V. Hernández, eds., 1999, pp. 411–424.
8. P. WESSELING, *An Introduction to Multigrid Methods*, John Wiley & Sons, New York, 1992.
9. D. XIE AND L. R. SCOTT, *The parallel u-cycle multigrid method*, in Proceedings of the 8th Copper Mountain Conference on Multigrid Methods, 1996.

PLIERS: A Parallel Information Retrieval System Using MPI

A. MacFarlane^{1,2}, J.A. McCann¹, S.E. Robertson^{1,2}

¹ School of Informatics, City University, London EC1V 0HB

² Microsoft Research Ltd, Cambridge CB2 3NH

Abstract. The use of MPI in implementing algorithms for Parallel Information Retrieval Systems is outlined. We include descriptions on methods for Indexing, Search and Update of Inverted Indexes as well as a method for Information Filtering. In Indexing we describe both local build and distributed build methods. Our description of Document Search includes that for Term Weighting, Boolean, Proximity and Passage Retrieval Operations. Document Update issues are centred on how partitioning methods are supported. We describe the implementation of term selection algorithms for Information Filtering and finally work in progress is outlined.

1. Introduction

We describe how MPI and the facilities that it provides are used to implement Parallel Information Retrieval Systems. Development work has been done at City University and further work is being continued at Microsoft Research in Cambridge on the PLIERS (ParaLLeL Information rEtrieval System) on which our description is based. A particular interest has been the production of a Parallel IR system which is portable: much of the previous work in the area has produced methods and systems which cannot be ported between different architectures [1]. One of the main reasons MPI was chosen as the mechanism for Message Passing was that it provided this facility. Another was that collective operations such as broadcast, Scatter and Gather are very useful in a transaction processing context.

Much of the work done on PLIERS is either heavily influenced or based on work done on the Okapi system at City University [2]; a uni-processor based IR system. This includes methods for Indexing (discussed in section 3) methods for Document Search (section 4), Document Update (section 5) and Information Filtering (section 6). We also include a brief description of Inverted files and how they are organised in Parallel IR systems (section 2) and describe work currently in progress (section 7). A summary is given in section 8.

2. IR and Inverted File Organisations

IR or Information Retrieval is concerned with the delivery of relevant documents to a user. We restrict our discussion to textual data. Text Retrieval systems use Indexes in the form of Inverted files. Inverted files are typically split up into two main parts: a keyword/dictionary file and a Postings file (also known as the Inverted list): see figure 1 for a simplified example.



Dictionary File			Postings File		
Word	Postings	Ptr	Id	Freq	Position List
lamb	2		15	1	[1]
.....			09	7	[1,2,3,4,5,6,7]
.....	2		15	1	[1]
mary			09	6	[2,3,4,7,8,9]
...					

Figure 1 - An Example Inverted File

The dictionary file stores keywords found in the text collection together with number of documents in which the keyword occurs and a pointer to a list of document records in the Postings file. Each posting list may contain data on the positions of words for each document. Two main approaches have been proposed for distributing Inverted Indexes to disks: *DocId* and *TermId* partitioning [3]: figure 2 gives a simple example of them. The *DocId* approach partitions the Index by document assigning a document to a single disk, while *TermId* assigns a unique term to a disk. PLIERS supports both type of partitioning methods and we outline the significance of partitioning methods on parallel IR algorithms below. A Document Map which contains such information as the Document Id, its length and location on disk is also utilised.



Dictionary Files				Postings File		
File	Word	Postings	Ptr	Id	Freq	Position List
1	lamb	2		15	1	[1]
				09	7	[1,2,3,4,5,6,7]
2	mary	2		15	1	[1]
				09	6	[2,3,4,7,8,9]

Figure 2a - Inverted File Partitioning Methods -*TermId*


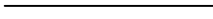

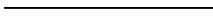
Dictionary Files				Postings File		
File	Word	Postings	Ptr	Id	Freq	Position List
1	lamb	1		15	1	[1]
	mary	1		15	1	[1]
2	lamb	1		09	7	[1,2,3,4,5,6,7]
	mary	1		09	6	[2,3,4,7,8,9]

Figure 2b - Inverted File Partitioning Methods -*DocId*

We use an architecture which is well known in the Parallel Systems field namely the Shared Nothing architecture. In essence this means each node in the parallel machine has its own CPU, main memory and local disk. Since the Index cannot be kept in memory for most applications (particularly web search engines) we use the Shared Nothing architecture to parallelise I/O as well as processing.

3. Indexing

Indexing is the generation of the Inverted file given the particular text collection. The indexing consists of parsing the text to identify words, recording their occurrence in the dictionary and updating posting/position data in the Inverted list. Two methods for the use of Parallel Indexing on text collections have been implemented: Local build (section 3.1) and Distributed build (section 3.2). We describe how these build methods relate to the partitioning method below.

3.1 Local Build Indexing

With Local Build the text collection is split up into N sub-collections and distributed to local disks to each node. The process of Indexing is quite simple: N Indexers are applied to each sub-collection in parallel that can proceed independently and very little communication is needed. In fact MPI is hardly used at all in the method apart from the initial start up messages and a MPI_Barrier command to notify a timing process that all Indexers have finished their work. Only *DocId* partitioning is available for this type of build as all data and text for any given document is kept locally.

3.2 Distributed Build Indexing

Distributed build indexing keeps the text collection on one disk using a Text Farmer process to distribute documents to a number of Worker Indexer processes which do much the same work as the Indexers in Local build. The method can distribute single documents or text files, but in practice we distribute text files because it reduces the level of communication needed. The communication interaction between the Farmer and Worker uses the MPI_Ssend, MPI_Irecv and MPI_Waitsome functions. The distributed indexing starts with the Farmer giving each worker an initial text file for analysis together with a given set of document identifiers: MPI_Ssend is used. The farmer then waits for the worker to request either another file to index or a new set of document id's. We use MPI_Waitsome to wait for request for work and if more than one request is received they are recorded in a worker queue at the Farmer, so that no data is lost. The farmer keeps a buffer of asynchronous receives issuing a MPI_Irecv for a worker each time it has completed servicing a request. From the point of view of the farmer the file interaction is a simple MPI_Irecv / MPI_Ssend interaction whereas the document id interaction requires the exchange of document map data: the interaction is therefore MPI_Irecv/MPI_Recv/MPI_Ssend. When the text collection has been distributed, the farmer sends a termination notice using MPI_Ssend to each worker. This has different implications for different request types. If the request type

is for more id's then a termination is not sent: the worker has further work to do. Termination is only sent to the worker when a File is requested. The last set of map data can then be received at the farmer. The termination interaction is identical to the file server interaction. It should be noted that some problems were found when using `MPI_Send` in the termination interaction: the worker moves into a phase which requires a great deal of CPU and I/O resources which prevented the send map data message being sent on one MPI implementation sequentialising the termination process: `MPI_Ssend` solved the problem.

If the *DocId* partitioning method is used then the process stops, but a further process is needed for *TermId* called the Global Merge. The Global Merge is split up into three parts: collection of partition statistics for allocation to nodes, distribution of data to nodes given one of a number of criteria on those statistics and a local merge to create the file Index for that node. Only the first two parts use MPI functions. The Statistics collection part uses `MPI_Gather` to obtain one of the following quantities: number of words in a partition, collection frequency in a partition and term frequency in a partition. A heuristic is then applied to the chosen quantity, which allocates partitions to nodes. The data is then distributed to nodes by gathering files in partitioning order to the nodes which has been allocated that particular partition: `MPI_Gather` is used.

4. Document Search

Document search in terms of Inverted files is the submission of user search requests in the form of a Query and applying it to the index using specified operations. These operations can be explicit, for example AND in boolean logic, or implicit as found in many web search engines which use term weighting operations. Search in PLIERS is based on the Search Set method, which in essence works by applying merge operations to sets of postings taken from the Inverted File. Parallel search has two types of processes: a Top or interface process that communicates with the client and collates data, and a number of leaf processes that manage a given Inverted file fragment. The interaction between the Top and Leaf processes is as follows: the top process receives a query and broadcasts it (using `MPI_Bcast`) to the leaf processes; the leaf processes retrieve and merge sets, sending only as much data to the top as is needed using a gathering mechanism (`MPI_Gather` is used). The last point has particular significance for the partitioning method used and we discuss this for Boolean/Proximity operations (section 4.1), Term Weighting Operations (section 4.2), and Passage Retrieval (section 4.3).

4.1 Boolean and Proximity Operations

Boolean operations on search sets use the normal Union (OR), Intersection (AND) and Difference (AND NOT) methods found in set theory. Proximity operations are an extension on Boolean operators and provide a further restriction on search e.g. searching for two words which are adjacent to each other (message ADJ passing). Proximity operations use position data that may have the format field, paragraph, sentence and word positions.

DocId. Boolean or Proximity operations of any type can be applied to each set element because all data on a document is kept local. The Leaf result sets can then be gathered by the Top process which does a final OR merge to produce the result ready for presentation at the client.

TermId. For some simple operations such as OR the set merge and process interaction is identical to *DocId* partitioning query service. However in many cases the final result cannot be computed until all data has been gathered e.g. all Proximity operations. This means that all search sets must be transmitted to the Top process for it to do all the set merges: parallelism is therefore restricted on this type of partitioning. It should be noted that unlike *DocId*, in *TermId* some Leaf nodes may have no work to do if a given query has no keywords in that partition: this has implications for load balance which affects all search set operations.

4.2 Term Weighting Operations

Users apply these operations by specifying a natural language query e.g. “parallel text retrieval”. Term Weighting operations assign a weight to a keyword/document pair and do a set merge in order to calculate a total score for each document. Term Weighting operations have following phases: retrieve sets for the keyword, weight all sets given collection statistics, merge the sets accumulating scores for documents and sort for final results. With both types of partitioning method, set retrieve can be done in parallel, but different strategies are needed for the other phases. It should be noted that map data statistics must be exchanged on Local build Indexes: using MPI_Reduce followed by MPI_Bcast does this. This communication is not needed for distributed build as the map data is replicated.

DocId. Certain collection statistics such as collection frequency for a keyword do not reside on one node. Therefore the top process must gather this data and the information recorded in the query to be sent back to the Leaf process. Currently MPI_Gather is used, but alternatively MPI_Reduce could be used or MPI_Allreduce that would restrict message exchange between leafs. When weighting is done local set merges can be done in parallel and sorts initiated on local set results. The top N results are then picked off by each Leaf and sent to the Top node: MPI_Gather is used. This method restricts the amount of data to be transmitted to the Top node. The final result is generated by picking of the top set off all leaf results based on descending weight order. This result can then be delivered to the client.

TermId. Term weighting can proceed in parallel without any communication and set merges can be applied in parallel to generate the result set for a Leaf. However sorts must be done at the top node, as the weight for any given document has not been generated. The Top process (MPI_Gather is used) therefore gathers the intermediate result sets and a merge is applied to generate the final result set. The sort can then be applied and the top set of results picked off ready for presentation to the client. More

communication is needed in this method, and no parallelism is available for one of the most important aspects of Weighting operations, namely the sort.

4.3 Passage Retrieval

Passage Retrieval search is the identification of a part of a document which may be relevant to a user e.g. in a multiple subject document. A computationally intensive algorithm for Passage Retrieval has been implemented [4] that is of order $O(n^3)$ unoptimised (this is reduced to $O(n^2)$ using various techniques). The algorithm works by iterating through a contiguous list of text atoms (paragraphs have been used) and applying a Term Weighting function to each passage, recording the best weighted passage. We outline two methods on *DocId* partitioning for this type of search: a Term Weighting operation is applied and Passage retrieval is only done on the top set of results.

Method 1. With this method, the Passage Retrieval algorithm is applied to the top set of results locally at each node. Currently this is the top 1000 documents on each node. Therefore many more documents are examined than in the next strategy. A second sort is applied at each node to produce a final result set and the same operation described in *DocId* weighting search is used to generate the final result set.

Method 2. This method applies the Passage Retrieval algorithm only on the top 1000 documents found in Weighting search over the whole collection. We use a document accumulator mechanism for this. The top set identified during weighted search is broadcast (using `MPI_Bcast`) and each Leaf node generates Passage Retrieval scores for its documents. The results are then gathered by the Top node (using `MPI_Gather`) and a final sort is applied to generate the final result.

5. Document Update

Document update is the addition of new documents to an existing Index. We reuse code from the Indexing module but use it in a Index maintenance context. The update of Inverted Indexes is very resource intensive because of the need to keep individual Inverted Lists in contiguous storage to enable efficient search: I/O is a significant cost in IR systems. We have hypothesized that parallelism may reduce Update costs and have implemented a method on both types of partitioning. The search topology is used for update: this allows for both search and update transaction service. Updates and search requests are delivered to the system. New documents are recorded in a buffer and the Leaf nodes reach distributed agreement on a re-organisation of the whole Indexing using `MPI_Allreduce` if one node has reached its buffer limit: when this condition is met all further transactions are locked out until the Index has been reorganised. We are currently considering various aspects of this algorithm.

6. Information Filtering

Information Filtering is the process of supplying documents to a user based on a long term information need: in this application the Query is persistent (though it may be modified). Documents chosen by the user as being relevant are examined and a set of terms is chosen using a statistical technique. In [4] it was stated that an alternative to some ranking methods described, would be to "evaluate every possible combination of terms on a training set and used some performance evaluation measure to determine which combination is best". Various heuristics have been implemented for Term Selection in order to examine some of this search space. Given that these algorithms can take many hours or days we have applied parallel techniques to them in order to speed up processing. Since a Term can be evaluated independently we can split up the Term Set amongst processes and each node can apply the chosen Term Selection algorithm in parallel to its given sub-set of terms in the training set. Terms are scattered to Slave nodes by the Master node (MPI_Scatter is used). The Index is replicated so any node can apply the algorithms on any term. The Term Selection algorithms are applied iteratively until one of a number of stopping criteria is reached. After each iteration MPI_Gather is used to obtain the best term for that iteration, and nodes are notified of the choice using MPI_Bcast. Performance data is gathered using MPI_Gather when the Term Selection algorithm is complete.

7. Work in Progress

We discuss various aspects of our work with MPI including ease of portability, programming with MPI with its advantages/disadvantages and performance.

7.1 Ease of Portability

Various implementations of MPI have been used by PLIERS including CHIMP [5], MPICH [6] and ANU/Fujitsu MPI [7]. We did find some differences between implementations such as different type for MPI_comm (which is an *int* in some systems). Various bugs in some of the implementations also caused some problems. Using MPI (with GNU C) has allowed us to port our software to different types of architectures such as a Network of Workstations (NOWS), the Fujitsu AP1000 and AP3000 parallel machines and an Alpha Farm. We have completed a port to a Cluster of 16 PC's connected by a supercomputing interconnect running the Windows NT operating system: MPI was invaluable in this process.

7.2 Programming with MPI

Our experience with using different MPI implementations and architectures has been positive. We have found the rank system a useful abstraction particularly when used with collective operations: maintaining code is made easier than other methods such as OCCAM-2 (any topology change would require the re-write of hard wired collective operations). MPI is much more flexible. However this flexibility has its price. The requirement that implementation can vary part of the message passing semantics to

cope with lack of buffering space led directly to the termination problem in indexing described above. There is a fairly large set of routines and ideas to learn in order to use MPI to the full, much more so than OCCAM-2. We have not used PVM so cannot compare it with MPI, but we would use MPI rather than OCCAM-2.

7.3 Performance

Results on the architectures currently supported are being generated and examined, but early indications show that there is a performance gain to be had on IR systems by using parallelism. Currently PLIERS can index 14/15 Gigabytes of raw text per hour on 8 Alpha nodes if no position data is needed and term weighting search shows near linear speedup. We will report these and other results in more detail at a later date.

8. Summary

We have found that MPI is a useful method for implementing portable and efficient parallel Information Retrieval systems. In particular we have found that many collective operations are useful for both Query transaction service in IR and Term Selection in Information Filtering. Other message passing facilities have been found to be useful however. Our experience of using different implementations of MPI on different architectures has been positive.

References

1. MacFarlane, A., Robertson, S.E., McCann, J.A.: Parallel Computing in Information Retrieval - An updated Review. *Journal of Documentation*. Vol. 53 No. 3 (1997) 274-315
2. S. E. Robertson: Overview of the OKAPI projects. *Journal of Documentation*. Vol. 53 No. 1 (1997) 3-7
3. Jeong, B., Omiecinski, E.: Inverted file partitioning schemes in multiple disk systems. *IEEE Transactions on Parallel and Distributed Systems*. Vol. 6 No. 2 (1995) 142-153
4. Robertson, S.E., Walker, S., Jones, S., Hancock-Beaulieu, M.M. Gatford, M.: Okapi at TREC-3. In: Harman, D.K. (ed): *Proceedings of Third Text Retrieval Conference*, Gaithersburg, USA (1994) 109-126
5. Alasdair, R., Bruce, A., Mills J.G., Smith, A.G.: CHIMP/MPI User Guide Version 1.2, EPCC-KTP-CHIMP-V2-USER1.2. Edinburgh Parallel Computing Centre (1994)
6. Gropp, W., Lusk, E.: Users guide for MPICH, a portable Implementation of MPI. Mathematics and Computer Science Division, Argonne National Laboratory, University of Chicago (1998)
7. MPI User's Guide. ANU/Fujitsu CAP Research Program, Department of Computer Science, Australian National University (1994)

Parallel DSIR Text Retrieval System

Arnon Rungsawang, Athichat Tangpong, Pawat Laohawee
{fenganr,g4165239,g4165221}@ku.ac.th

KU Text REtrieval Group (KU-TREG)
Department of Computer Engineering
Faculty of Engineering
Kasetsart University, Bangkok, Thailand

Abstract. We present a study concerning the applicability of a distributed computing technique to a million-page free-text document retrieval problem. We propose a high-performance DSIR retrieval algorithm on a Beowulf PC Pentium cluster using PVM message-passing library. DSIR is a vector space based retrieval model in which semantic similarity between documents and queries is characterized by semantic vectors derived from the document collection. Retrieval of relevant answers is then interpreted in terms of computing the geometric proximity between a large number of document vectors and query vectors in a semantic vector space. We test this DSIR parallel algorithm and present the experimental results using a large-scale TREC-7 collection and investigate both computing performance and problem size scalability issue.

1 Introduction

Among the grand challenge applications in high performance computing, free-text retrieval is of particular interest in this information explosion epoch. While the number of new web pages on the Internet, as well as free-text documents in scientific, academic, business, and industrial usages, increase without limit day by day, seeking the relevant information in these million-page documents needs rapid and powerful searching artifact to get through. We believe that recent advances in distributed processing techniques on cluster of workstations or PCs [3, 2] is currently mature enough and can provide powerful computing means convenient for overcoming this grand challenge application.

In this paper we focus on using cluster computing technique to implement a parallel free-text retrieval algorithm called “DSIR” [5]. DSIR is a vector space based retrieval model in which semantic similarity between documents is characterized by semantic vectors derived from the document collection. Documents and queries whose vectors are closed in a semantic space are assumed to have similar or related contexts. Since DSIR is a quite complex text retrieval system designed for Information Retrieval research usage in our laboratory, indexing and searching time with a large-scale text collection is inevitably much longer than general search engine found in the Internet. To speed up the retrieval process, we thus propose a parallel DSIR retrieval algorithm using PVM message-passing

library [1] on a cluster of low-cost Beowulf PC Pentium class machines running Linux operating system [7].

Our Beowulf PC cluster is composed of several Pentium-II 350MHz, 128MB of physical memory machines, interconnected by two-way 10 Mbps and 100 Mbps Ethernet routing networks. Both NFS and message-passing traffics can be configured to pass through these two-way routes in a convenient way depending on the application (i.e. heavy NFS or heavy message-passing traffics) which is running on. In our retrieval experiments, we choose TREC-7 documents [8] as our large test collection. TREC-7 collection consists of more than half-million and varying length free-text documents, 2 GB in total. We index the TREC-7 collection and represent each document with a 500-dimensional vector. During retrieval, that large number of document vectors are then compared with standard TREC-7 50 query vectors using our parallel DSIR algorithm. We also test the problem size scalability issue by duplicating the TREC-7 collection two times and increasing the workload from 50 to 300 queries to search with. We found that our algorithm still scale quite well with this size of collection.

We organize this paper in the following way. Section 2 briefly presents the DSIR retrieval model. Section 3 discusses the proposed parallel algorithm. Section 4 gives more detail about our experimental setup, results, and discussion. Finally, section 5 concludes this paper.

2 DSIR Retrieval Model

In DSIR retrieval model, the distributional semantic information inherently stored in the document collection is utilized to represent document contents in order to achieve better retrieval effectiveness [4]. The distributional information used in our current system implementation is characterized by co-occurrence vectors of words selected as index terms for a particular document collection. Both documents and queries input to DSIR are then defined as the weighted vector sum of the co-occurrence vectors corresponding to index terms occurring in those documents and queries.

As documents and queries are represented as vectors in the same vector space, the basic retrieval operation is then very simple; the query vector is compared to every document vector, documents whose vectors locate in the vicinity of that query vector are presented to the user in decreasing order of their closeness as relevant answer. A typical vector similarity measure that we use is either the cosine similarity or dot-product function [6].

Fig. 1 depicts a simple DSIR document indexing and retrieval diagram. First, documents are pre-processed (i.e. eliminating stopwords and performing word stemming) and then put into co-occurrence computational and document statistical extraction routines. Both co-occurrence information and document statistics are then used to compute all document vectors. Here we suppose that query vectors are derived by the same method as documents. Each query vector is searched against document vectors for its related documents in document vector database using similarity computational routine. Output of this routine will be

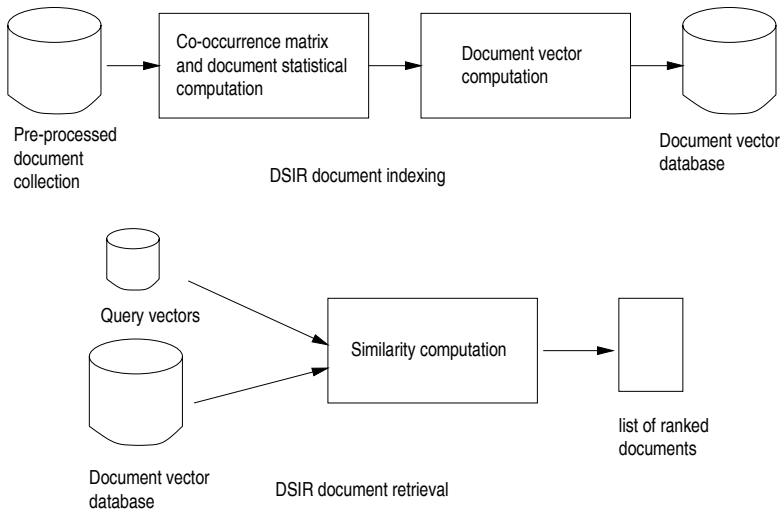


Fig. 1. DSIR Retrieval Model.

a list of ranked documents in decreasing order of similarity with respect to the input query.

3 Parallel DSIR Retrieval Algorithm

We follow master-slave programming model to develop our parallel retrieval algorithm [1]. The algorithm consists of four main steps; partitioning the document vector database to distribute to other computing nodes, sending query vectors to computing nodes, computing vector similarity, and rescoring all similarity scores (see Fig. 2).

We can explain this parallel DSIR retrieval algorithm in more detail as follows; First, a stripping algorithm is responsible for dividing a large document vector database into several chunks. The size of each chunk is depending on how computing nodes can allocate and perform similarity computation without performing local disk paging. Second, the first chunks of document vectors are sent to corresponding computing nodes while in this case master behaves like a file-server. Third, to optimize the message-passing traffic, master sends all query vectors in advance to all computing nodes. Fourth, each computing node computes similarity scores between document vectors and all query vectors, and sends similarity scores back to master for final rescoring. Fifth, if there are still any chunks of document vectors left, master distributes next chunks of document vectors to computing nodes, and loops back to step four above. Sixth, master recomputes the final ranking scores and outputs the list of ranked documents as relevant answer.

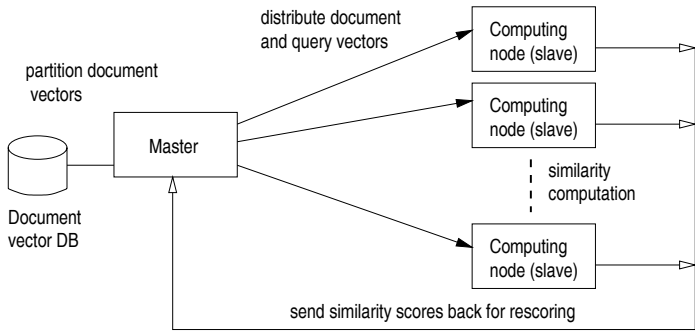


Fig. 2. Parallel DSIR retrieval diagram.

4 Experimental Results

In this section, we present the results concluding from two sets of experiments. For the first experimental setup, we study the computing performance between “via-NFS retrieval” and “distributed database retrieval” (see Fig. 3). For document retrieval via-NFS, we share the single large document vector database to other computing nodes via 100 Mbps Ethernet switch. For distributed database retrieval, we split the same database into several portions and store those portions to the local disk of the computing nodes¹. For the second experimental setup, we examine the scalability issue of our implementation by doubling the number of documents in the collection and varying the number of input queries.

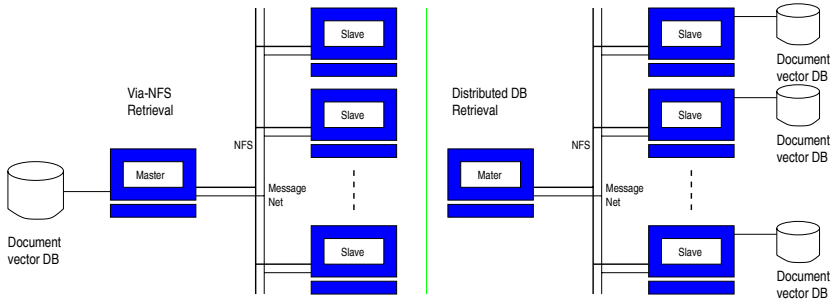


Fig. 3. Via-NFS vs. Distributed Database Retrieval.

¹ Evidently we can see that via-NFS retrieval configuration is less expensive to set up than distributed database retrieval one as it is not necessary to have a harddisk installed at every computing node, and all computing nodes can be set up as diskless nodes.

4.1 Via-NFS vs. Distributed Database Retrieval

We prepare 528,155 TREC-7 documents to be our test database [8]. Each document is indexed and represented with 503 dimensional vector so that the total document vector database yields around 1 GB. The original 50 TREC-7 queries are indexed and represented by 503 dimensional vectors as well. Retrieval algorithm now can then be interpreted as computing the similarity scores (e.g. vector scalar product) between each query vector and every document vector in 1 GB database, and ranking that 528,155 similarity scores. Several curves from Fig. 4 concludes the results from these experiments.

As we have expected, experimental results show that via-NFS retrieval setup is less effective than distributed database retrieval one. Via-NFS retrieval setup gives longer average query response time, lower CPU utilization, and less efficiency. When the number of computing nodes increase from one to two, average CPU utilization decreases sharply from 85.15% to 19.61%, the average query response time increases from 19.98 to 30.74 seconds, and speedup factor decreases almost 50%. We think that the main reason of this performance degradation comes from the fact that during document retrieval each computing node spends almost of its time to wait for document vectors from master via NFS, though that NFS has been routed via an 100 Mbps Ethernet switch (i.e. NFS bottle-neck).

For the distributed database retrieval, each computing node can load up the corresponding document vectors directly from its local disk. When the number of computing nodes increase from one to two, CPU utilization drops from 79% to 62%, and average query response time slowly decreases. But speedup factor increases when the number of computing nodes increase till 5 nodes, and slightly decreases beyond that point.

After examining closely what the problem occurs with the computing performance when adding more than 5 computing nodes on distributed database retrieval setup, we discover two problems. The first one comes from the fact that we inadvertently let every computing node writes back its similarity scores to master for final rescoring via 10 Mbps NFS route. Thus when more computing nodes have been added, more intense NFS bottle-neck occurs. The second one comes from the problem of load unbalancing in our document vector database splitting algorithm. Some computing nodes that have less documents to perform similarity computation will finish their computing faster than the others, and then stay idle. We then correct these two problems by rerouting final similarity scores via 100 Mbps line, fixing the splitting algorithm, and rerunning our tests. Fig. 5 which depicts average query response time and speedup curves supports this finding. Note that we obtain a nearly perfect speedup curve.

4.2 Scalability Issue

To study the scalability issue of the proposed algorithm, we set up the experiments in the following way. We simply double the old TREC-7 collection two times. This new database finally yields more than one million documents, and

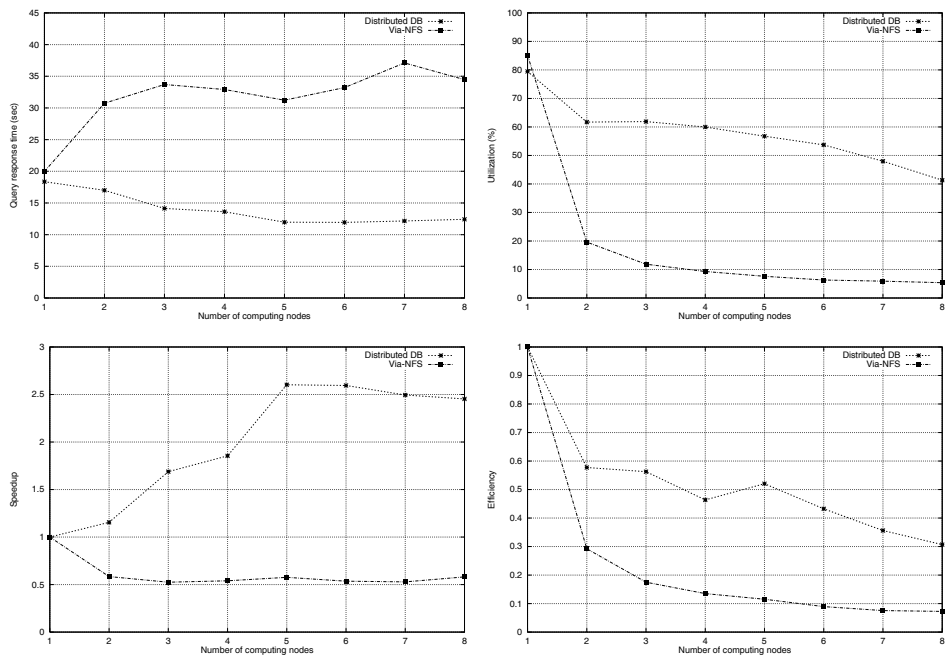


Fig. 4. Via-NFS vs. Distributed database retrieval.

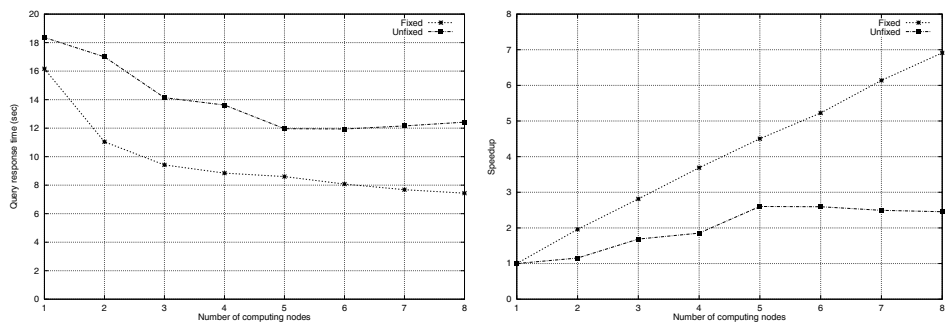


Fig. 5. Distributed database retrieval after fixing the problem of load unbalancing.

is around 2 GB of document vectors. We also build the second set of queries by duplicating the old query set 6 times, and rerun more experiments using distributed database retrieval setup. Now, the old 50-query set and the new 300-query set are searched against a million-page document vectors. Results from these experiments have been depicted in Fig. 6.

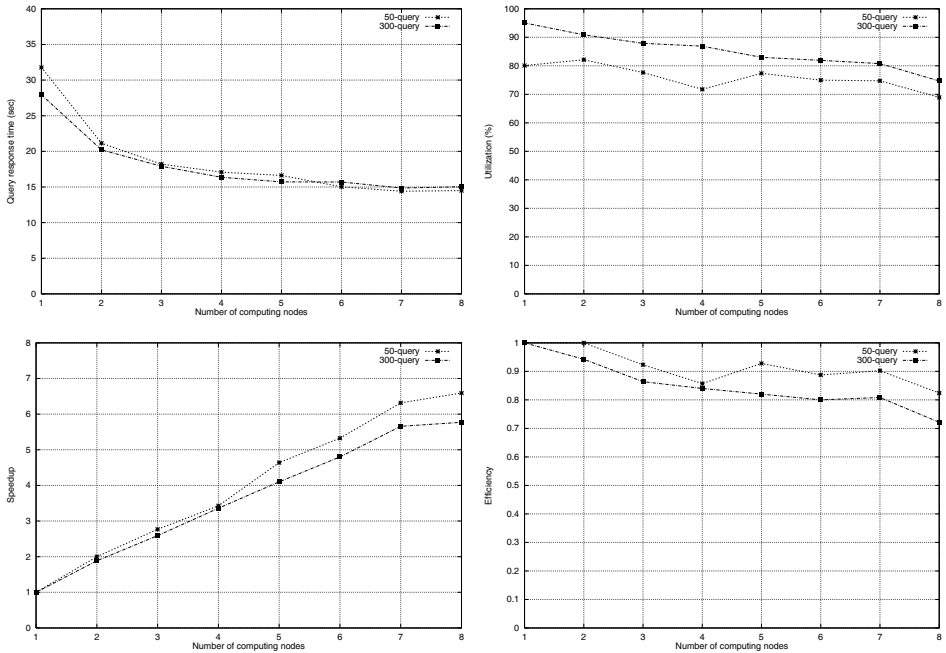


Fig. 6. Problem size scalability when doubling the old document database and increasing the number of input queries.

The results show that our proposed algorithm still scales well though the problem size has been increased two times (i.e. doubling the document vector database), and the work load has been increased six times (i.e. increasing the number of queries to 300). Average query response times similarly decrease to 15 seconds when the number of computing nodes have been added to 8. CPU utilization of 300-query set is higher than another, due to the fact that the timing expense of very long similarity computation which happens in parallel at every computing node compensates much more than the timing expense of rescoring the similarity scores which is the only sequential part in our current implementation. Moreover, speedup factor also linearly increases when the computing nodes are increased.

5 Conclusion

In this paper we have proposed a parallel DSIR text retrieval algorithm, and presented our experimental results using a large TREC-7 document collection. We have learned from this study that applying the distributed computing technique on a low-cost Beowulf class PC cluster is an efficient way to solve a grand challenge problem like large-scale free-text retrieval application.

We have also discovered two main problems that degrade the computing performance when using message-passing technique on this kind of PC cluster. The first one comes from the classical problem of NFS bottle-neck. Since a free-text retrieval is an I/O intensive application in general, partitioning a large retrieval problem into smaller chunks and distributing them to other computing nodes via NFS, though it is less expensive to set up, is very inefficient to do. The second problem comes from the problem of load unbalancing. When a large problem size has not been well divided, some computing nodes will be sometimes left idle if they have less workload to compute.

From our experiments, we also found that our proposed algorithm scales well though we increase our problem size to a million-page documents and double the input query workload from one to six times (i.e. from 50 to 300 queries). However, as we know well, just million-page documents are not a real problem size for an acceptable search engine in the Internet anymore, we thus anticipate to ameliorating our parallel DSIR retrieval algorithm and testing it with several million web-page documents in the near future.

References

1. A. Geist et al. PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing. The MIT Press, 1994.
2. J. Dongarra et al. Integrated PVM Framework Supports Heterogeneous Networking Computing. *Computers in Physics*, 7(2):166–175, April 1993.
3. T.E. Anderson et al. A Case for NOWs. *IEEE Micro*, February 1995.
4. A. Rungsawang. DSIR: The First TREC-7 Attempt. In E. Voorhees and D.K. Harman, editors, *Proceedings of the Seventh Text REtrieval Conference*. NIST Special publication, November 1988.
5. A. Rungsawang and M. Rajman. Textual Information Retrieval Based on the Concept of the Distributional Semantics. In *Proceedings of the 3th International Conference on Statistical Analysis of Textual Data*, December 1995.
6. G. Salton and M.J. McGill. Introduction to Modern Information Retrieval. McGraw Hill, 1983.
7. P. Uthayopas. Beowulf Class Cluster: Opportunities and Approach in Thailand. In *First NASA workshop on Beowulf class computer systems*. NASA JPL, October 1997.
8. E.M. Voorhees and D.K. Harman. Overview of the Seventh Text REtrieval Conference (TREC-7). In *Proceedings of the Seventh Text REtrieval Conference*. NIST Special publication, November 1998.

PVM Implementation of Heterogeneous ScaLAPACK Dense Linear Solvers

Vincent Boudet, Fabrice Rastello, and Yves Robert

LIP, UMR CNRS-ENS Lyon-INRIA 5668

Ecole Normale Supérieure de Lyon

F - 69364 Lyon Cedex 07

`Vincent.Boudet@ens-lyon.fr`

Abstract. This paper discusses some algorithmic issues when computing with a heterogeneous network of workstations (the typical poor man's parallel computer). How is it possible to efficiently implement numerical linear algebra kernels like those included in the ScaLAPACK library ? Dealing with processors of different speeds requires to use more involved strategies than purely static block-cyclic data distributions. Dynamic data distribution is a first possibility but may prove impractical and not scalable due to communication and control overhead. Static data distributions tuned to balance execution times constitute another possibility but may prove inefficient due to variations in the processor speeds (e.g. because of different workloads during the computation). There is a challenge in determining a trade-off between the data distribution parameters and the process spawning and possible migration (redistribution) policies. We introduce a semi-static distribution strategy that can be refined on the fly, and we show that it is well-suited to parallelizing several kernels of the ScaLAPACK library such as LU and QR decompositions.

Keywords: Heterogeneous networks, distributed-memory, different-speed processors, scheduling and mapping, numerical libraries.

1 Introduction

Heterogeneous networks of workstations are ubiquitous in university departments and companies. They represent the typical poor man's parallel computer: running a large PVM or MPI experiment (possibly all night long) is a cheap alternative to buying supercomputer hours. The idea is to make use of *all* available resources, namely slower machines *in addition to* more recent ones.

The major limitation to programming heterogeneous platforms arises from the additional difficulty of balancing the load when using processors running at different speed. In this paper, we explore several possibilities to implement linear algebra kernels on heterogeneous networks of workstations (NOWs). Our goal is to come up with a first proposal for an heterogeneous "Cluster ScaLAPACK" library devoted to dense linear system solvers: how to efficiently implement numerical kernels like LU or QR decompositions on an heterogeneous NOW?

Consider a heterogeneous NOW: whereas programming a large application made up of several loosely-coupled tasks can be performed rather easily (because these tasks can be dispatched dynamically on the available processors), implementing a tightly-coupled algorithm (such as a linear system solver) requires carefully tuned scheduling and mapping strategies. Distributing the computations (together with the associated data) can be performed either dynamically or statically, or a mixture of both. At first sight, we may think that dynamic strategies are likely to perform better, because the machine loads will be self-regulated, hence self-balanced, if processors pick up new tasks just as they terminate their current computation. However, data dependences, communication costs and control overhead may well lead to slow the whole process down to the pace of the slowest processors. On the other hand, static strategies will suppress (or at least minimize) data redistributions and control overhead during execution. To be successful, static strategies must obey a more refined model than standard block-cyclic distributions: such distributions are well-suited to processors of equal speed but would lead to a great load imbalance with processors of different speed. The design of static strategies that achieve a good load balance on a heterogeneous NOW is one of the major achievements of this paper.

2 Static Distribution

In this section we investigate static strategies for implementing ScaLAPACK routines on heterogeneous NOWs. Static strategies are less general than dynamic ones but constitute an efficient alternative for heavily constrained problems. The basis for such strategies is to distribute computations to processors so that the workload is evenly balanced, and so that no processor is kept idle by data dependences. We start with the simple problem of distributing independent chunks of computations to processors, and we propose an optimal solution for that problem in Section 2.1. We use this result to tackle the implementation of linear solvers. We propose an optimal data distribution in Section 2.2.

2.1 Distributing Independent Chunks

To illustrate the static approach, consider the following simple problem: given M independent chunks of computations, each of equal size (i.e. each requiring the same amount of work), how can we assign these chunks to p physical processors P_1, P_2, \dots, P_p of respective execution times t_1, t_2, \dots, t_p , so that the workload is best balanced? Here the execution time is understood as the number of time units needed to perform one chunk of computation. A difficulty arises when stating the problem: how accurate are the estimated processor speeds? won't they change during program execution? We come back on estimating processor speeds later, and we assume for a while that each processor P_i will indeed execute each computation chunk within t_i time units. Then how to distribute chunks to processors? The intuition is that the load of P_i should be inversely proportional to t_i . Since the loads must be integers, we use the following algorithm to solve the problem:

Algorithm 2.1: Optimal distribution for M independent chunks, over p processors of speed t_1, \dots, t_p

```
# Approximate the  $c_i$  so that  $c_i \times t_i \approx \text{Constant}$ , and  $c_1 + c_2 + \dots + c_p \leq M$ .
forall  $i \in \{1, \dots, p\}$ ,  $c_i = \left\lfloor \frac{\frac{1}{t_i}}{\sum_{i=1}^p \frac{1}{t_i}} \times M \right\rfloor$ .
# Iteratively increment some  $c_i$  until  $c_1 + c_2 + \dots + c_p = M$ 
for  $m = c_1 + c_2 + \dots + c_p$  to  $M$ 
    find  $k \in \{1, \dots, p\}$  such that  $t_k \times (c_k + 1) = \min\{t_i \times (c_i + 1)\}$ 
     $c_k = c_k + 1$ 
```

Proposition 1. *Algorithm 2.1 gives the optimal allocation.*

Complexity and Use. Since, $c_1 + c_2 + \dots + c_p \geq M - p$, there are at most p steps of incrementation, so that the complexity of Algorithm 2.1 is¹ $O(p^2)$. This algorithm can only be applied to simple load balancing problems such as matrix-matrix product on a processor ring.

When processor speeds are accurately known and guaranteed not to change during program execution, the previous approach provides the best possible load balancing of the processors. Let us discuss the relevance of both hypotheses:

Estimating Processor Speed. There are too many parameters to accurately predict the actual speed of a machine for a given program, even assuming that the machine load will remain the same throughout the computation. Cycle-times must be understood as *normalized cycle-times* [4], i.e application-dependent elemental computation times, which are to be computed via small-scale experiments (repeated several times, with an averaging of the results).

Changes in the Machine Load. Even during the night, the load of a machine may suddenly and dramatically change because a new job has just been started. The only possible strategy is to “use past to predict future”: we can compute performance histograms during the current computation, these lead to new estimates of the t_i , which we use for the next allocation. See the survey paper of Berman [1] for further details.

In a word, a possible approach is to slice the total work into phases. We use small-scale experiments to compute a first estimation of the t_i , and we allocate chunks according to these values for the first phase. During the first phase we measure the actual performance of each machine. At the end of the phase we collect the new values of the t_i , and we use these values to allocate chunks during the second phase, and so on. Of course a phase must be long enough, say a couple of seconds, so that the overhead due to the communication at the end of each phase is negligible. Each phase corresponds to B chunks, where B is chosen by the user as a trade-off: the larger B , the more even the predicted load, but the larger the inaccuracy of the speed estimation.

¹ Using a naive implementation. The complexity can be reduced down to $O(p \log(p))$ using ad-hoc data structures.

2.2 Linear Solvers

Whereas the previous solution is well-suited to matrix multiplication, it does not perform efficiently for LU decomposition. Roughly speaking, the LU decomposition algorithm works as follows for a heterogeneous NOW: blocks of r columns are distributed to processors in a cyclic fashion. This is a *CYCLIC*(r) distribution of columns, where r is typically chosen as $r = 32$ or $r = 64$ [2]. At each step, the processor that owns the pivot block factors it and broadcasts it to all the processors, which update their remaining column blocks. At the next step, the next block of r columns become the pivot panel, and the computation progresses. The preferred distribution for a homogeneous NOW is a *CYCLIC*(r) distribution of columns, where r is typically chosen as $r = 32$ or $r = 64$.

Because the largest fraction of the work takes place in the update, we would like to load-balance the work so that the update is best balanced. Consider the first step. After the factorization of the first block, all updates are independent chunks: here a chunk consists of the update of a single block of r columns. If the matrix size is $n = M \times r$, there are $M - 1$ chunks. We can use Algorithm 2.1 to distribute these independent chunks.

But the size of the matrix shrinks as the computation goes on. At the second step, the number of blocks to update is only $M - 2$. If we want to distribute these chunks independently of the first step, redistribution of data will have to take place between the two steps, and this will incur a lot of communications. Rather, we search for a static allocation of columns blocks to processors that will remain the same throughout the computations, as the elimination progresses. We aim at balancing the updates of all steps with the same allocation. So we need a distribution that is kind of repetitive (because the matrix shrinks) but not fully cyclic (because processors have different speeds). Looking closer at the successive updates, we see that only column blocks of index $i + 1$ to M are updated at step i . Hence our objective is to find a distribution such that for each $i \in \{2, \dots, M\}$, the amount of blocks in $\{i, \dots, M\}$ owned by a given processor is approximately inversely proportional to its speed. To derive such a distribution, we use a dynamic programming algorithm which is best explained using the former toy example again:

A Dynamic Programming Algorithm. In Table 1, we report the allocations found by the algorithm up to $B = 10$. The entry “Selected processor” denotes the rank of the processor chosen to build the next allocation. At each step, “Selected processor” is computed so that the cost of the allocation is minimized. The cost of the allocation is computed as follows: the execution time, for an allocation $\mathcal{C} = (c_1, c_2, \dots, c_p)$ is $\max_{1 \leq i \leq p} c_i t_i$ (the maximum is taken over all processor execution times), so that the average cost to execute one chunk is

$$\text{cost}(\mathcal{C}) = \frac{\max_{1 \leq i \leq p} c_i t_i}{\sum_{i=1}^p c_i}$$

For instance at step 4, i.e. to allocate a fourth chunk, we start from the solution for three chunks, i.e. $(c_1, c_2, c_3) = (2, 1, 0)$. Which processor P_i should

receive the fourth chunk, i.e. which c_i should be incremented? There are three possibilities $(c_1 + 1, c_2, c_3) = (3, 1, 0)$, $(c_1, c_2 + 1, c_3) = (2, 2, 0)$ and $(c_1, c_2, c_3 + 1) = (2, 1, 1)$ of respective costs $\frac{9}{4}$ (P_1 is the slowest), $\frac{10}{4}$ (P_2 is the slowest), and $\frac{8}{4}$ (P_3 is the slowest). Hence we select $i = 3$ and we retain the solution $(c_1, c_2, c_3) = (2, 1, 1)$.

Table 1. Running the dynamic programming algorithm with 3 processors: $t_1 = 3$, $t_2 = 5$, and $t_3 = 8$.

Number of chunks	c_1	c_2	c_3	Cost	Selected processor
0	0	0	0		1
1	1	0	0	3	2
2	1	1	0	2.5	1
3	2	1	0	2	3
4	2	1	1	2	1
5	3	1	1	1.8	2
6	3	2	1	1.67	1
7	4	2	1	1.71	1
8	5	2	1	1.87	2
9	5	3	1	1.67	3
10	5	3	2	1.6	

Proposition 2. (see [3]) *The dynamic programming algorithm returns the optimal allocation for any number of chunks up to B .*

The complexity of the dynamic programming algorithm is $O(pB)$, where p is the number of processors and B , the upper bound on the number of chunks. Note that the cost of the allocations is not a decreasing function of B .

Application to LU Decomposition. For LU decomposition we allocate slices of B blocks to processors. B is a parameter that will be discussed below. For a matrix of size $n = m \times r$, we can simply let $B = m$, i.e. define a single slice.

Within each slice, we use the dynamic programming algorithm for $s = 0$ to $s = B$ in a “reverse” order. Consider a toy example with 3 processors of relative speed $t_1 = 3$, $t_2 = 5$ and $t_3 = 8$. The dynamic programming algorithm allocates chunks to processors as shown in Table 2. The allocation of chunks to processors is obtained by reading the second line of Table 2 from right to left: $(3, 2, 1, 1, 2, 1, 3, 1, 2, 1)$. At a given step there are several slices of at most B chunks, and the number of chunks in the first slice decreases as the computation progresses (the leftmost chunk in a slice is computed first and then there only remains $B - 1$ chunks in the slice, and so on). In the example, the reversed allocation best balances the update in the first slice at each step: at the first step when there are the initial 10 chunks (1 factor and 9 updates), but also at the second step when only 8 updates remain, and so on. The updating of the other slices remains well-balanced by construction, since their size does

not change, and we keep the best allocation for $B = 10$. See Figure 1 for the detailed allocation within a slice, together with the cost of the updates.

Table 2. Static allocation for $B = 10$ chunks.

Chunk number	1	2	3	4	5	6	7	8	9	10
Processor number	1	2	1	3	1	2	1	1	2	3

2.3 A Proposal for a Cluster ScaLAPACK

We are ready to propose a first solution for an heterogeneous cluster ScaLAPACK library devoted to dense linear solvers such as LU or QR factorizations. It turns out that all these solvers share the same computation unit, namely the processing of a block of r columns at a given step. They all exhibit the same control graph: the computation processes by steps; at each step the pivot block is processed, and then it is broadcast to update the remaining blocks.

The proposed solution is fully static: at the beginning of the computation, we distribute slices of the matrix to processors in a cyclic fashion. Each slice is composed of B chunks (blocks of r columns) and is allocated according to the previous discussion. The value of B is defined by the user and can be chosen as M if $n = m \times r$, i.e. we define a single slice for the whole matrix. But we can also choose a value independent of the matrix size: we may look for a fixed value, chosen from the relative processor speeds, to ensure a good load-balancing.

A major advantage of a fully static distribution with a fixed parameter B is that we can use the current ScaLAPACK release with little programming effort. In the homogeneous case with p processors, we use a *CYCLIC*(r) distribution for the matrix data, and we define p PVM processes. In the heterogeneous case, we still use a *CYCLIC*(r) distribution for the data, but we define B PVM processes which we allocate to the p physical processors according to our load-balancing strategy. The experiments reported in the next section fully demonstrate that this approach is quite satisfactory in practice.

3 PVM Experiments

In this section we report several PVM experiments that fully demonstrate the usefulness of the static approach explained in Section 2.2 .

Description. In this section, we report experiments on an heterogeneous NOWs made of 6 different workstations of the LIP laboratory, interconnected with Ethernet/IP (we call this network *lip*). We compare the ScaLAPACK implementation of the standard purely cyclic allocation (*CYCLIC*(r) to be precise) with a PVM implementation of our static distribution (with $B = 9$). We use

two different matrix decomposition algorithms, namely LU and QR. Matrices are of size $n \times n$, and they are divided into blocks of $r = 32$ columns.

As already pointed out, these experiments have been obtained with little programming effort, because we did not modify anything in the ScaLAPACK routines. We only declared several PVM processes per machine. We did pay a high overhead and memory increase for managing these processes. This is a limitation in the use of our program as such: we chose the value $B = 9$ instead of $B = \frac{n}{r}$, which would have led to a better allocation. A refined implementation (maybe using MPI kernels) would probably yield better results.

We mentioned in Section 2.1 that processor cycle-times are to be computed via small-scale experiments (repeated several times, with an averaging of the results). Hence, for each algorithm, we have measured the computation time on different matrix sizes.

Now, we investigate the value of the “reasonable” speedups that should be expected. We use the example of LU decomposition on the *lip* network. Processor speeds are (100,161,284,297,303,326). One can say that on an heterogeneous NOW, asymptotically, the computation time for LU decomposition with cyclic distribution is imposed by the slowest processor. Hence, in the example, a *cyclic*(6) distribution of column blocks to processors will lead to the same execution time as if computed with 6 identical processors of speed $\frac{326}{6} = 54$. Let q be the slowest processor, i.e. suppose that $\forall i, t_q \geq t_i$. The best computation time that can be achieved on the heterogeneous NOW would be $t_{optimal} \approx \frac{1}{\sum_{i=1}^p \frac{1}{t_i}} \times \frac{p}{t_q} \times t_{cyclic(p)}$. One also can approximate the computation time of our distribution by $t_{hetero(B)} \approx \frac{\max_{i=1}^p c_i t_i}{B} \times \frac{p}{t_q} \times t_{cyclic(p)} = t_{theoretical}$.

Experiments. For each algorithm (LU, QR), we represent on the same graph the execution time for:

- **Cyclic Distribution:** It is the ScaLAPACK/PVM implementation.
- **Our Distribution:** 9 processes are created on different processors according to the distribution given by the dynamic programming algorithm. Then we use the ScaLAPACK/PVM implementation with a *CYCLIC*(32) distribution onto the 9 processes.
- **Optimal Algorithm:** It is a theoretical computation time, proportional to the *cyclic distribution* computation time. The ratio is calculated as previously explained.

The effects of cache size, and of the pivoting computations make our algorithm faster than the theoretical approximation does predict.

4 Conclusion

In this paper, we have discussed static allocation strategies to implement dense linear system solvers on heterogeneous computing platforms. Such platforms are likely to play an important role in the future. We have shown both theoretically and experimentally (through PVM experiments) that our data and computation distribution algorithms were quite satisfactory.

We also have compared the static approach with dynamic solutions, and we have show that we could reach comparable (or even better) performances, while retaining the simplicity of the implementation.

The next project would be to target a collection of heterogeneous NOWs rather than a single one. Implementing linear algebra kernels on several collections of workstations or parallel servers, scattered all around the world and connected through fast but non-dedicated links, would give rise to a “Computational Grid ScaLAPACK”. Our results constitute a first step towards achieving this ambitious goal.

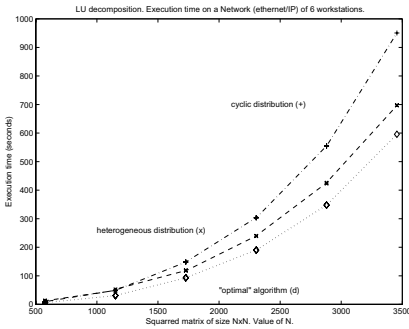


Fig. 1. For LU decomposition on *lip*, the ratio $t_{cyclic(6)}$ over $t_{optimal}$ is 1.6 and $\frac{t_{cyclic(6)}}{t_{theoretical}} = 1.5$.

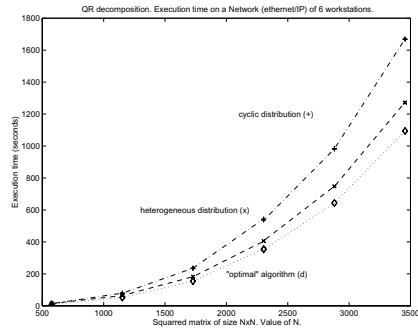


Fig. 2. For QR decomposition on *lip*, the ratio $t_{cyclic(6)}$ over $t_{optimal}$ is 1.5 and $\frac{t_{cyclic(6)}}{t_{theoretical}} = 1.4$.

References

1. F. Berman. High-performance schedulers. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 279–309. Morgan-Kaufmann, 1998.
2. L. S. Blackford, J. Choi, A. Cleary, E. D’Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users’ Guide*. SIAM, 1997.
3. P. Boulet, J. Dongarra, F. Rastello, Y. Robert, and F. Vivien. Algorithmic issues for heterogeneous computing platforms. Technical Report RR-98-49, LIP, ENS Lyon, 1998. Available at www.ens-lyon.fr/LIP.
4. Michal Cierniak, Mohammed J. Zaki, and Wei Li. Scheduling algorithms for heterogeneous network of workstations. *The Computer Journal*, 40(6):356–372, 1997.

Using PMD to Parallel-Solve Large-Scale Navier-Stokes Equations. Performance Analysis on SGI/CRAY-T3E Machine

Jalel Chergui

Institut du Dveloppement et des
Ressources en Informatique Scientifique.
CNRS/IDRIS, Bt. 506, BP167
F-91406 Orsay cedex, France
Jalel.Chergui@idris.fr

Abstract. PMD (Parallel Multi-domain Decomposition) is an MPI based Fortran 90 module which objective is to parallel-solve positive definite linear elliptic second order equations. It has been used to solve unsteady Navier-Stokes equations in order to simulate an axisymmetric incompressible viscous fluid flow inside a centrifugal pump. In this paper, we will present a brief description of the implementation and will discuss the performance measurements obtained on the SGI/CRAY T3E parallel machine.

1 Introduction

PMD [CHE 98] is an MPI [GRO 96] based Fortran 90 module which was developed at IDRIS¹ in cooperation with LIMSI². It's a public domain software³ which objective is to parallel-solve elliptic linear second order equations. This arises the question: How PMD can be used to solve parabolic non-linear time-dependent systems as the unsteady Navier-Stokes equations ?

The answer, which will be discussed in this paper, consists in applying a classical time-discretization procedure which transforms the continuous time-dependent parabolic non-linear equations into a time-discretized elliptic linear systems. In turn, this arises the choice of the time-discretization scheme which has to be consistent with the continuous equations and to ensure an acceptable accuracy when advancing the flow in time. In the next paragraphs, we will review the Navier-Stokes equations and establish the time-procedure used in this study to discretize these equations.

¹ Institut du Dveloppement et des Ressources en Informatique Scientifique, CNRS/IDRIS, Bt. 506, BP167, F-91406 Orsay cedex, France.

<http://www.idris.fr>

² Laboratoire d'Informatique et de Mcanique pour les Sciences de l'Ingénieur, CNRS/LIMSI, UPR 3251, BP133, F-91406 Orsay cedex, France.

<http://www.limsi.fr>

³ can be retrieved following this link:

<http://www.idris.fr/data/publications/PMD/PMD.html>

1.1 Mathematical Model

We shall assume an axisymmetric flow of a newtonian incompressible viscous fluid. The Navier-Stokes equations are formulated in terms of stream function ψ , azimuthal component ω of the vorticity and tangential component v of the velocity. This leads to the following set of equations formulated in a cylindrical coordinate system $(r, z) \in [R_{in}, R_{out}] \times [0, 1]$ (figure 1):

$$\frac{\partial \omega}{\partial t} - \overbrace{\frac{1}{R_e} \left(\nabla^2 - \frac{I}{r^2} \right) \omega}^{\text{Diffusive terms}} = \underbrace{-\frac{\partial(u\omega)}{\partial r} - \frac{\partial(w\omega)}{\partial z} + \frac{1}{r} \frac{\partial v^2}{\partial z}}_{\text{Convective terms}} \quad (1)$$

$$\frac{\partial v}{\partial t} - \overbrace{\frac{1}{R_e} \left(\nabla^2 - \frac{I}{r^2} \right) v}^{\text{Diffusive terms}} = \underbrace{-\frac{\partial(uv)}{\partial r} - \frac{\partial(wv)}{\partial z} - \frac{2uv}{r}}_{\text{Convective terms}} \quad (2)$$

$$\left(\frac{\partial^2}{\partial z^2} + r \frac{\partial}{\partial r} \left(\frac{1}{r} \frac{\partial}{\partial r} \right) \right) \psi = r\omega \quad (3)$$

Where u and w are respectively the radial and axial components of the velocity. ∇^2 and I are the Laplace and the Identity operators respectively. $R_e = \frac{\Omega H^2}{\nu}$ is the Reynolds number (based on the angular velocity Ω of the rotor, on the distance H between the two disks and on the kinetic viscosity ν) which is the only free dimensionless parameter in this problem.

All physical boundary conditions are of DIRICHLET type and stem from the adherence of the fluid particles to the walls.

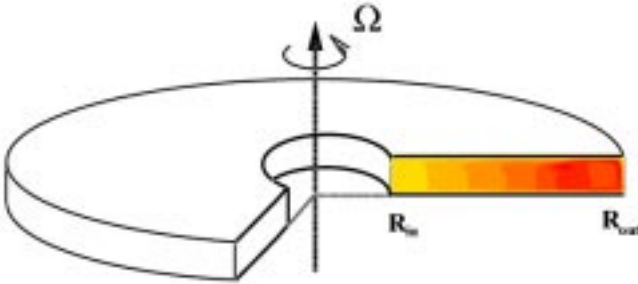


Fig. 1: The computational domain. The interdisk domain shows the mean stream function field at $R_e = 5000$, $P_r = 0.72$, $R_{in} = 2$, $R_{out} = 8$ and $\frac{H}{R_{out}} = 0.125$. Courtesy of RMI JACQUES, UMR6600, Universit  de Technologie de Compigne, BP 20529, 60200 Compigne, France

1.2 Numerical Model

The equations 1 and 2 have been discretized using a semi-implicit second order finite difference scheme where the convective terms are discretized using an ADAMS-BASHFORTH procedure while the CRANK-NICOLSON scheme has been applied to discretize the diffusive terms. In order to clarify this point, let's consider the following advection-diffusion equation:

$$\frac{\partial f}{\partial t} + \underbrace{V \nabla f}_{\text{convective term}} = \underbrace{\nabla^2 f}_{\text{diffusive term}} \quad (4)$$

where f is any scalar field and V is the transport velocity. Let's denote by Δt the timestep and by n the index of the timestep. ADAMS-BASHFORTH time-discretization is applied on the convective term while the diffusive term is discretized using the CRANK-NICOLSON scheme. This procedure leads to the following second order time-discretized equation:

$$\frac{f^{n+1} - f^n}{\Delta t} + \frac{3}{2}(V \nabla f)^n - \frac{1}{2}(V \nabla f)^{n-1} = \frac{1}{2}(\nabla^2 f^{n+1} + \nabla^2 f^n) \quad (5)$$

The previous equation leads to the well-known HELMHOLTZ problem which reads:

$$(\sigma I - \nabla^2) f^{n+1} = S_f^n \quad (6)$$

where S_f^n is the source term which includes all known quantities computed at the timestep n while seeking the solution at the timestep $n + 1$.

This discretization procedure, applied to the equations 1 and 2, yields a set of coupled elliptic linear equations:

$$\left(\left(\sigma + \frac{1}{r^2} \right) I - \nabla^2 \right) \omega^{n+1} = S_{\omega, v, \psi}^n \quad (7)$$

$$\left(\left(\sigma + \frac{1}{r^2} \right) I - \nabla^2 \right) v^{n+1} = S_{v, \psi}^n \quad (8)$$

$$\left(\frac{\partial^2}{\partial z^2} + r \frac{\partial}{\partial r} \left(\frac{1}{r} \frac{\partial}{\partial r} \right) \right) \psi^{n+1} = r \omega^{n+1} \quad (9)$$

where $\sigma = \frac{2R_e}{\Delta t}$. At each timestep $n + 1$, two HELMHOLTZ problems (7, 8) have to be solved together with the stream-function equation (9).

2 Application Development

The objective was to develop a message passing parallel application in a short timeframe (2 days) that efficiently solves the problem defined by equations 1, 2 and 3. The PMD module (release 1.0.2) has been used since it offers all the

building box to quickly parallel-solve such a problem. For each operator, PMD applies a domain decomposition based on the Schur complement ([QUA 90]) to build the Schur matrix and to solve the subdomain interface problem. Figure 2 summarizes the PMD routines calling sequence to reach the final solution.

As we notice, the system defined by equations 7, 8 and 9 is composed of two elliptic HELMHOLTZ problems and one elliptic Stream-Function equation. Moreover, if we compare equations 7 and 8, we notice that both equations exhibit the same HELMHOLTZ operator $\left(\sigma + \frac{1}{r^2}\right) I - \nabla^2$. Hence, two Schur matrices have to be build (figure 2: lines 24, 25). The former corresponds to the HELMHOLTZ operator and the latter to the Stream-Function one. During this step, each process solves a homogeneous time-independent problem to build its own block of the Schur matrix which is then LU-factored (figure 2: lines 29, 30). Conversely, in the time-loop, each process seeks time-dependent final local solutions of ω , v and ψ fields for which we parallel-solve three equations (figure 2: lines 37-39) at each timestep iteration.

Furthermore, standard second order centered finite differences have been used for both first and second order spatial derivatives. Subsequently, each process solves a local problem of size $(N_r \times N_z)$ using fast direct solvers. These methods are documented as FOURIER-TOEPLITZ or FOURIER-tridiagonal methods [LAS 87]. In our case, the procedure mainly consists in locally performing a diagonalization in the z -direction using FFT routines and in performing a finite differences method in the r -direction.

3 Performance

Timing measurements and performance rates of the application will be presented in terms of sustained 64 bit floating point operations per second (Mflops/sec) on the SGI/CRAY T3E-600 parallel machine whose characteristics and compiler options are summarized in Table 1. The instrumentation has been performed

Table 1: Hardware and software characteristics of the used T3E machine.

#Processors	256
Processor	Dec Alpha EV5
Processor memory	128 MB
Memory data cache	8 KB + 96 KB two level caches
Processor peak performance	600 MFlops/sec
Interconnexion network	Bidirectional 3D Torus: 600 MB/sec
Operating system	Unicos/mk 2.0
MPI Library	MPT (Version 1.2)
Compiler	f90 (Version 3.1)
Optimization options	-O3,unroll2,pipeline3

```

1  PROGRAM NS
2  USE PMD
3  !... Declare User and MPI type objects
4  ...
5  !... Extended PMD communicator
6  TYPE(PMD_Comm_2D) :: Comm
7  !... PMD datatype to handel the Schur matrices associated
8  !... to the Helmholtz and to the Stream-Function operators
9  TYPE(PMD_R8_Schur) :: H, SF
10 !... PMD datatype to handel the LU factorization
11 !... of the Schur matrices
12 TYPE(PMD_R8_GELU) :: LU_H, LU_SF
13
14 !... Begin PMD
15 CALL MPI_INIT(...)
16 CALL PMD_Init_1DD(..., Comm)
17
18 !... Set the local Helmholtz and Stream-Function
19 !... operator matrices and initial conditions
20 ...
21
22 !... Build the Schur matrices of the Helmholtz
23 !... and of the stream-function operators
24 CALL PMD_Schur_1DD(Comm, ..., H)
25 CALL PMD_Schur_1DD(Comm, ..., SF)
26
27 !... LU-Factor the Schur matrices of the Helmholtz
28 !... and of the stream-function operators
29 CALL PMD_Schur_Factor_1DD(Comm, H, LU_H)
30 CALL PMD_Schur_Factor_1DD(Comm, SF, LU_SF)
31
32 !... Start timestep loop
33 DO
34 !... Set time-dependent boundary conditions
35 ...
36 !... Solve the final problem for  $\omega$ ,  $v$  and  $\psi$  fields
37 CALL PMD_Solve_1DD(Comm, ..., LU_H, ..., omega)
38 CALL PMD_Solve_1DD(Comm, ..., LU_H, ..., v)
39 CALL PMD_Solve_1DD(Comm, ..., LU_SF, ..., psi)
40
41 !... print the results
42 IF(convergence) EXIT
43 END DO
44
45 !... End PMD
46 CALL PMD_End_1DD(Comm)
47 CALL MPI_FINALIZE(...)
48 END PROGRAM NS

```

Fig. 2: Fortran code to parallel-solve the Navier-Stokes equations using PMD.

with `MPI_WTIME` function and with the SGI/CRAY `apprentice` tool. Nevertheless, the total elapsed time returned by `MPI_WTIME` has always been compared to the **user processors connect time** reported by the `acctcom` CRAY command. This is in order to avoid instrumented cases corresponding to situations in which processes are held or checkpointed by the system job scheduler during the execution. Actually, in such situation, the elapsed times returned by the `MPI_WTIME` function are over estimated with respect to the connect time.

3.1 Global Performance Analysis

Table 2 shows the elapsed time and the global performance rate on 64 processors of the T3E. The most cost effective part of the application is the resolution step during which we performed 5000 timestep iterations. Especially noteworthy is the time induced by the MPI communication calls which represents almost 21% of the total elapsed time on 64 processors. This percentage includes, however, MPI point-to-point and collective communication times. Since MPI collective communications are blocking routines, this measured time includes the overhead induced by all local MPI synchronization and internal buffer management operations.

Table 2: Elapsed time and performance rate on 64 processors. Mesh size in each subdomain: $N_r = 129$, $N_z = 81$. Timestep: $\Delta t = 0.00008$. Reynolds Number: $Re = 1000$. Radius of inner and outer cylinders: $R_{int} = 1$, $R_{out} = 11$.

Time to Build the Schur Matrices	180. sec.
Time to LU-Factor the Schur Matrices	90. sec.
Time to Solve the Problem: 5000 timestep iterations	6600. sec.
Total Elapsed Time	6939. sec.
Total MPI Communication Time	1470. sec.
Global Performance Rate on 64 processors	2.7 GFlops/sec.

More details are obtained using the CRAY `apprentice` performance tool. Figure 3 represents a snapshot section of the `apprentice` window. It shows the global cost of the MPI communication overhead (dark colored bar) with respect to the cost of the parallel work which, as we can see, is induced mainly by the ScaLAPACK routine `PSGETRS` called inside the PMD module.

3.2 Scalability

In this case study, the subdomain mesh size is maintained constant for any processors number. The ideal scalable parallel application would be the one with no inter-process communications. Such kind of application should return a constant total elapsed execution time per meshpoint and per timestep iteration for any processors number. However, in a realistic context of a parallel application, the communication rate may has some influence on the scalability of the code beyond a critical number of processors.

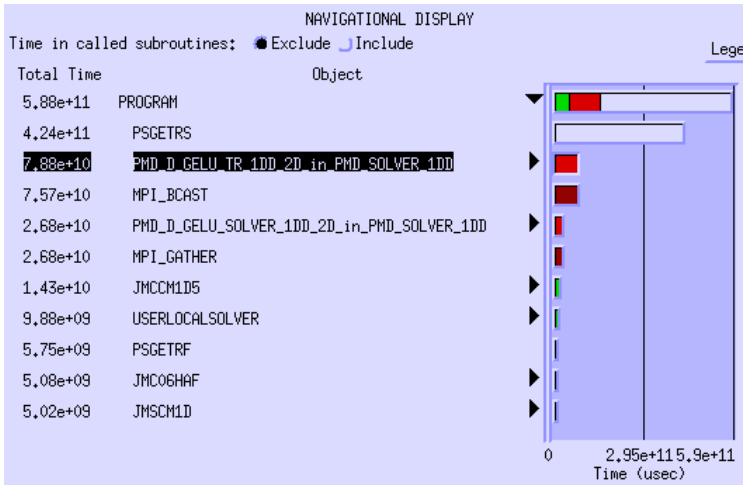


Fig. 3: Apprentice window section. Details of the global parallel work. Dark colored bars mostly represent MPI communication overheads. Processors number: 64. Mesh size in each subdomain: $N_r = 129$, $N_z = 81$. Timestep: $\Delta t = 0.00008$. Reynolds Number: $R_e = 1000$. Radius of inner and outer cylinders: $R_{int} = 1$, $R_{out} = 11$.

In fact, the application scales quiet well till 64 processors (figure 4). Beyond this limit, the number of processors being larger, the communication rate has greater incidence over the computational part. Especially noteworthy is that till 16 processors the elapsed time rather decreases. This has to be related to a better reuse of the memory data cache at the ScaLAPACK routine level [CHER 97] since the Schur matrices have been block-distributed among the processors. Hence beyond 16 processors, the block size of the Schur matrices seems to be better appropriate.

Also, it has to be noted (figure 4) that the local memory allocation size grows linearly as the processors number increases. Hence on 100 processors the amount of allocated memory is only 2.7 times larger than on 25 processors, though the global mesh size is 4 times larger on 100 processors.

4 Conclusion

We used PMD release 1.0.2 to develop a message passing parallel code within a short timeframe to solve the Navier-Stokes equations. The application simulates an axisymmetric incompressible viscous fluid flow inside a centrifugal pump. It has run on the SGI/CRAY T3E-600 parallel machine at IDRIS and has delivered a total performance rate up to 6 GFlops/second on 144 processors. Performance and timing analysis have shown good scalability till 64 processors beyond which the MPI communication overhead has taken larger influence with respect to the processor computational load.

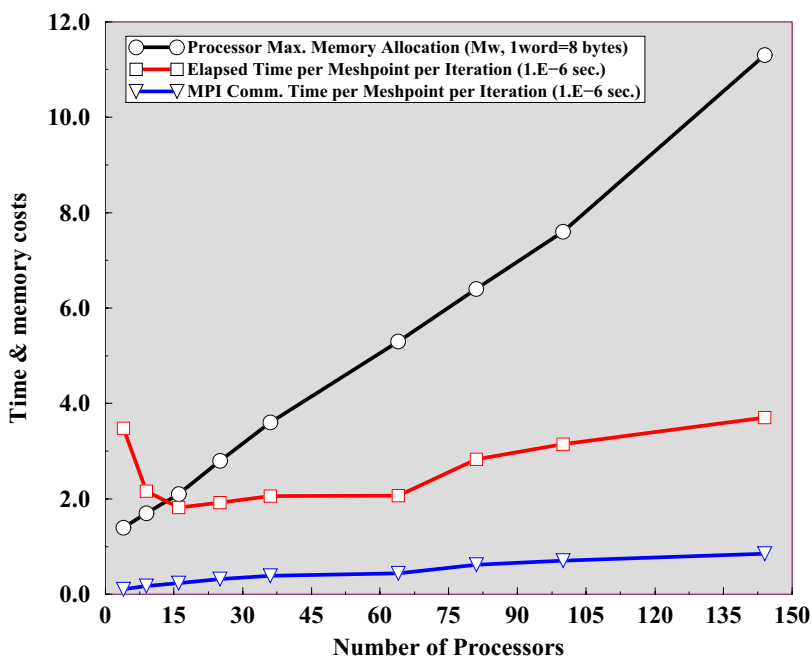


Fig. 4: Scalability of the application on the T3E-600. Mesh size in each subdomain: $N_r = 129$, $N_z = 81$. Timestep: $\Delta t \in [0.0001, 0.00008]$. Reynolds Number: $Re = 1000$. Radius of inner and outer cylinders: $R_{int} = 1$, $R_{out} = 11$.

References

- CHE 98. J. CHERGUI & al., *PMD: A Parallel Fortran 90 Module to Solve Elliptic Linear Second Order Equations*. *Calculateurs Parallèles*, Hermes publisher, 75004 Paris, France. Vol. 10, N° 6, Dec. 1998.
- CHER 97. J. CHERGUI, Performance ratio analysis between CRAY T3E and T3D machines (in French). *Calculateurs Parallèles*, Hermes publisher, 75004 Paris, France. Vol. 9. No 4. December 1997.
- GRO 96. W. GROPP, S. HUSS-LEDERMAN, A. LUMSDAINE, E. LUSK, B. NITZBERG, W. SAFIR, M. SNIR, *MPI: The Complete Reference*. Volume 1, The MPI Core. The MIT Press Cambridge, Massachusetts London, England. 1998.
- JAC 98. R. JACQUES, P. LE QUR, O. DAUBE, *Parallel Numerical Simulation of Turbulent Flows in Closed Rotor-Stator Cavities*. Notes on Numerical Fluid Mechanics, Vieweg Publisher. 1998
- LAS 87. P. LASCAUX, R. THÉODOR, *Analyse numérique matricielle appliquée l'art de l'ingénieur*. Tome2, éd. Masson, Paris, France. 1987.
- QUA 90. A. QUARTERONI, *Domain Decomposition Method for the Numerical Solution of Partial Differential Equations*. UMSI 90/246. University of Minnesota Supercomputer Institute. Research report. December 1990.

Implementation Issues of Computational Fluid Dynamics Algorithms on Parallel Computers

Joanna Płazek¹, Krzysztof Banaś¹, Jacek Kitowski^{2,3}

¹ Section of Applied Mathematics UCK, Cracow University of Technology,
ul. Warszawska 24, 31-155 Cracow, Poland

² Institute of Computer Science, AGH, al. Mickiewicza 30, 30-059 Cracow, Poland

³ ACC CYFRONET-AGH, ul. Nawojki 11, 30-950 Cracow, Poland

Abstract. In this paper we compare efficiency of three versions of a parallel algorithm for finite element compressible fluid flow simulations on unstructured grids. The first version is based on the explicit model of parallel programming with message-passing paradigm, the second is the implicit model in which data-parallel programming is used, while in the last we propose a heterogeneous model which combines these two models.

Time discretization of the compressible Euler equations is organized with a linear, implicit version of the Taylor-Galerkin time scheme, with finite elements employed for space discretization of one step problems. The resulting nonsymmetric system of linear equations is solved iteratively with the preconditioned GMRES method.

The algorithm has been tested on two parallel computers HP SPP1600 and HP S2000 using a benchmark problem of 2D inviscid flow simulations – the ramp problem.

1 Introduction

Advent of new parallel computer architectures and development of compiler technology recapitulate implementations of parallel Computational Fluid Dynamics (CFD) algorithms. The most popular parallel implementations have been made using the explicit programming model (with message-passing programming paradigm). For experienced user it is possible to obtain high parallel efficiency with this model, often treated as an assembler for parallel computing. Another, more programmer's friendly model (characteristics for SMP processing) – with implicit programming (incorporating data-parallel paradigm) – is getting more interest at present. None of the models is universal; some integration of multiprocessing and multithreading would be profitable in the future.

The most popular, modern Cache Coherent Non-Uniform Memory Access (cc-NUMA), parallel computers are constructed with SMP nodes and offer Distributed Shared Memory (DSM) programming paradigm. The SMP nodes are interconnected in different ways. It can be a ring or 2D mesh connection (with SCI protocol) (e.g. HP Exemplar computers), multistage switching network (IBM RS6000 SP) or hypercube topology (SGI Origin2000). SMP node architecture

can also be different. One of the most sophisticated, applied in HP V2500 technical server, incorporates 8×8 crossbar with 4 PA8500 processors at each crossbar interface. Other machines use the bus technology (IBM RS6000 SP High Node) or the network technology with hubs and routers (like in SGI Origin2000 nodes).

In respect to variety of computer architectures different parallel algorithms for the finite element method (FEM) have been proposed (see e.g. [1, 2] and articles in [3, 4]). In the present paper we propose a general approach, taking into account a CFD problem to be solved in parallel as a whole.

Exploiting full potential of any parallel architecture requires a cooperative effort between the user and the computer system. The most profitable way is to construct the algorithm mapping well on the architecture. The domain decomposition should have, in general, large granularity of parallel tasks, which could result in load imbalance between processors, reducing benefits of parallel computing. SMP nodes with low latency and high bandwidth are more appropriate for the implicit programming. Combination of both paradigms defines a new one – heterogeneous programming paradigm – in which the implicit programming is applied within the SMP nodes while the message-passing paradigm between the nodes [5].

In the paper we present timing results for a CFD problem obtained on two kinds of computers: HP SPP1600 and HP S2000.

2 CFD Algorithm

The CFD algorithm is a time stepping procedure combined with a stabilized finite element formulation for space discretization of one step problems. The discretized equations are the Euler equations of compressible inviscid fluid flow in the conservation form. A time stepping procedure is a modified Taylor-Galerkin method consisting of a sequence of linear one step problems. It is used for transient problems as well as for driving solutions to the steady state, in case of stationary flows. At each time step a stabilized finite element problem is solved. A version of the GMRES method preconditioned by block Jacobi iterations is used to solve the system of linear equations resulting from standard finite element procedures. The algorithm uses the concept of patches of elements in the finite element mesh, that define blocks within the stiffness matrix. Only these blocks are assembled during the solution procedure, overcoming the problem of distributed storage of the global stiffness matrix. Matrix-vector products in the preconditioned GMRES algorithm are performed by means of loops over patches of elements and solutions of local block problems.

3 Parallel Implementation of GMRES

3.1 Parallelization with Explicit Programming Model

The GMRES algorithm preconditioned by the standard iterative methods can be interpreted as domain decomposition algorithm [6]. It reflects the general framework of Schwarz methods: divide the computational domain into subdomains,

solve the problem for each subdomain separately, combine the solutions. The role of subdomains is played by patches of elements and single iterations of iterative methods accelerated by the GMRES are just inexact solves of subdomain problems.

Since the number of patches could be big we divide the whole computational domain into subdomains which for the finite elements is just equivalent to mesh partitioning. The number of subdomains (submeshes) is equal to the number of computing nodes of a parallel machine. Since we use the distributed memory computational model, data corresponding to a particular subdomain are sent and stored in a memory of one computing node. The subdomains possess one element overlap so each node in the finite element mesh belongs to the interior of only one subdomain. The computing node corresponding to a given subdomain (storing its data) owns all its internal FE nodes and is responsible for all computations with these nodes.

Having distributed the data structure, computations of the GMRES algorithm begin. Some vector operations, such as scalar product or normalization, require communication between computing nodes and are done by standard message passing procedures using a selected computing node to gather the data. Other, like subtraction or multiplication by a scalar, are done perfectly in parallel. The GMRES minimization problem is solved on a chosen computing node.

Iterations of the block Jacobi method is done in two steps. First the loop over all internal FE nodes of corresponding subdomains is done by each computing node separately. Next, patches of elements are created and for each patch (i.e. finite element node) a local problem is solved by a direct solver (e.g. Gauss elimination).

Then all subdomains exchange data on interface nodes. Due to one element overlap between subdomains each node is updated during computations only by one computing node (corresponding to the subdomain owning the node). Hence the exchange of data after iteration loops is simplified – the computing node owning a given interface node just broadcast its data to neighboring subdomains.

3.2 Parallelization with Implicit Programming Model

Compilers for parallel computers parallelize codes automatically, verifying loop dependencies. Such an approach results in sufficient efficiency for simple loops only. For example, a subroutine call inside a loop prevents it from being parallelizable. To overcome those problems directives/pragmas are introduced to increase degree of parallelization and to control manually many aspects of execution.

In the implicit programming model the GMRES algorithm sketched above (written in C in a similar way to the explicit one) is parallelized using compiler directives whenever a loop over patches, nodes or individual degrees of freedom is encountered. In particular the parallelization is applied for:

- loops over blocks of unknowns at the step of blocks construction,
- computation of element stiffness matrices and assembly into patch stiffness matrices,
- iterations of the block method.

3.3 Parallelization with Hybrid Programming Model

A hybrid programming model is a combination of the explicit and implicit models. The idea behind is to have rather less but vast domains in order to reduce communication overhead and to minimize difficulties with load balancing and synchronization. Problems for each domain can be parallelized implicitly if one uses SMP nodes.

In this model, suitable for a machine with SMP nodes (or for a cluster of multiprocessor workstations), we introduce two levels of parallelization (for preliminary studies see [5, 7]):

- first-level parallelization with the explicit programming model making use of geometrical data decomposition into subdomains. A number of subdomains (submeshes) is equal to the number of SMP nodes coordinated with a message passing library.
- second-level parallelization with the implicit programming model for each SMP nodes, making use of the data-parallel paradigm for performing iterations for the block method.

4 Results

We have tested our parallel algorithm with an example of flow simulation where implicit finite element problems appear within the time discretization algorithm. The applied models are mentioned by PVM, DSM and PVM+DSM for the message-passing, the distributed shared memory and the hybrid models respectively.

We have chosen a well known transient benchmark problem of inviscid fluid flow – the ramp problem [8] – in which a Mach 10 shock traveling along a channel and perpendicular to its walls meets at time 0s a ramp having an angle of 60 degrees with the channel walls. The timing results referring to one time step are obtained by averaging over 5 subsequent time steps. Four numbers of nodes in the finite element mesh were employed, equal to 4474, 16858, 18241 and 73589 respectively.

We have used a version of the domain decomposition (mesh partition) algorithm that ensures vertical alignment of subdomain interfaces. The strategy results in the smallest number of the interface nodes minimizing the communication requirements. No mesh adaptation nor dynamic load balancing have been applied.

We have performed simulations on two HP Exemplar computers – SPP1600 and S2000. The first machine consists of 32 PA7200 processors organized in 4 SMP nodes (with software: SPP-UX 4.2, Convex C 6.5 and ConvexPVM 3.3.10).

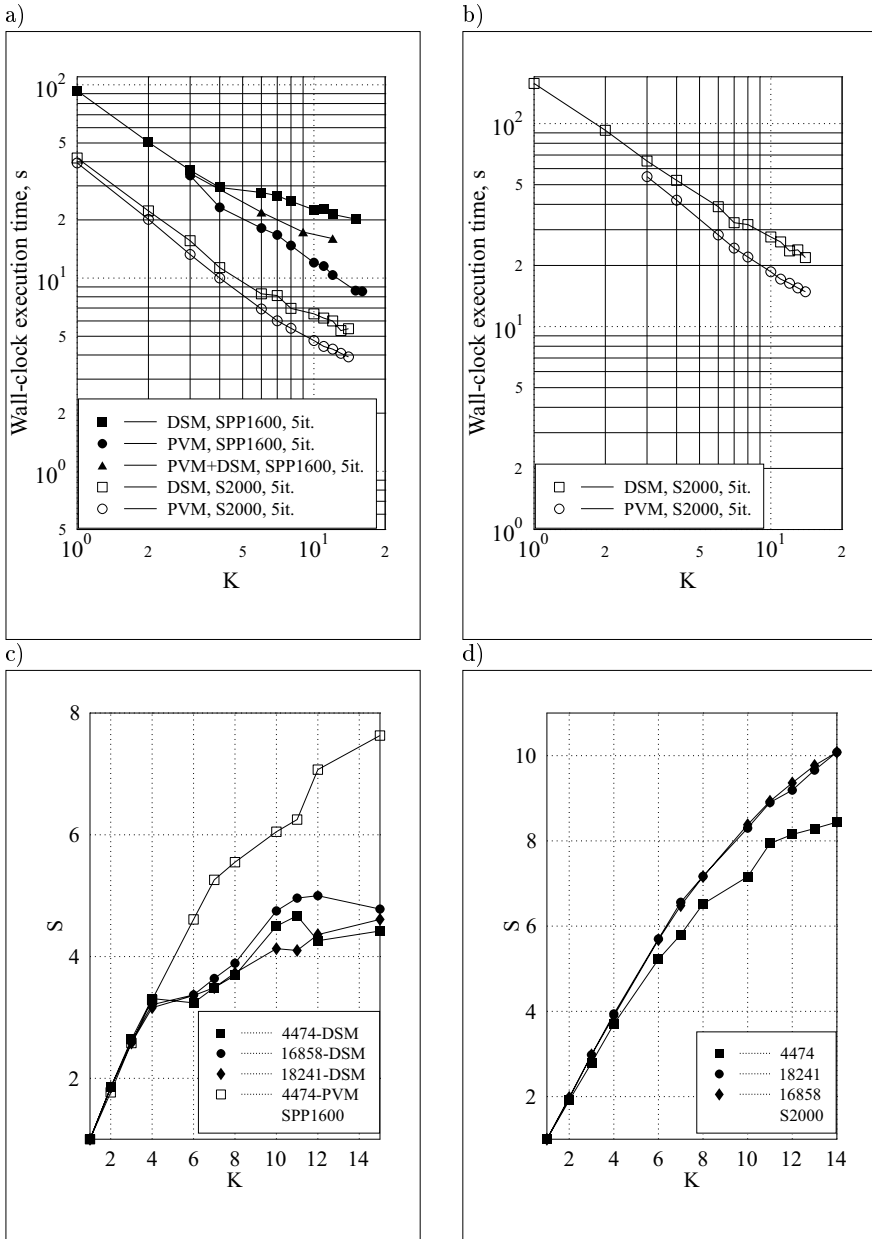


Fig. 1. Wall-clock execution time for one simulation step, different programming models and the mesh with (a) 18241 and (b) 73589 nodes; speedup for different number of nodes for (c) DSM and PVM model on SPP1600 and (d) PVM model on S2000.

A dedicated subcomplex consisting of 16 processors from four SMP (hyper-) nodes (4 processors from each hypernode at most) was used in the experiments. The second machine incorporates in total 16 PA8000 processors in one SMP node (using SPP-UX 5.2, HP C 2.1 and HP PVM 3.3.11). During the experiments no other users were allowed to use this machine.

In Fig.1a,b we present the wall-clock execution time for solution of one step problem using three (for SPP1600) or two (for S2000) programming models for different number of processors, K . Due to memory requirements the mesh with 73589 has been tested on S2000 only using PVM and DSM model with no hybrid model applied because the machine consists of one SMP node. Fig.1c,d represents results for speedup parameter, S , obtained for two extremes: for DSM model on SPP1600 (with one sample result concerning PVM) and for PVM model on S2000.

For SPP1600 and the explicit programming model efficiency is rather independent of the processing node organization – no substantial influence of the lower bandwidth between the SMP nodes is observed (see Fig.1c). For the implicit programming model worse scalability than for the PVM one (cf. Fig.1c,d) has been obtained, however this scalability is still within the acceptable range as long as one SMP node is concerned. Clear influence on scalability degradation is seen (Fig.1c) while adopting processors from another hypernode.

Due to better network layer organization, the characteristics obtained on S2000 are more smooth (in comparison with those of SPP1600 – see Fig.1d), maintaining the general relationship between explicit and implicit programming models.

For bigger problems using the explicit model (cf. Fig.1a,b) high demand for the local memory can be observed, so the program requires larger number of computing nodes in comparison with the implicit model. It means that we have been able to run the programs with PVM model for 4474 mesh nodes on SPP1600 and up to 18241 nodes on S2000 with the present memory configuration. In heterogeneous programming model, in principle, both kinds of the system memory – local and virtual shared – could be applied. Although not observed in this particular case, heterogeneous programming model could be applied for greater problems than the other models, making use of both memory kinds.

5 Conclusions

The implicit programming model can bring useful and scalable parallelization of FEM applications; it is more sensitive to communication speed than the explicit one. Explicit programming gives better results for the price of more complicated code structure and appropriate domain decomposition with load balancing considered. Otherwise, the synchronization would degrade potentially high parallel efficiency.

Changing from one SMP node execution to multi- SMP node execution only mildly affects the performance of the explicit code, while deteriorates execution time of the implicit code. Still we hope that implicit model is worth testing due

to the relative ease of programming and possible future improvements in compiler technology. On many-processor machines with hierarchical network layer architecture probably the heterogeneous model would be the choice, since efficiency degradation is not high in comparison with message-passing model, with better, in principle, memory utilization.

Acknowledgments

The work has been partially sponsored by Polish State Committee for Scientific Research (KBN) Grants No. 8 T11F 003 12 and 8T11C 006 15.

References

1. G.F.Carey, A.Bose, B.Davis, C.Harle, and R.McLay, Parallel computation of viscous flows, in *Proceedings of the 1997 Simulation Multiconference. High Performance Computing '97, 6-10 April 1997, Atlanta, USA*.
2. G.Mahinthakumar, F.Saied, and A.J.Valocchi, Comparison of some parallel Krylov solvers for large scale ground-water contaminant transport simulations, in *Proceedings of the 1997 Simulation Multiconference. High Performance Computing '97, 6-10 April 1997, Atlanta, USA*.
3. *Finite Element Method in Large-Scale Computational Fluid Dynamics, International Journal for Numerical Methods in Fluids*, **21**, (1995).
4. Idelsohn, S.R., Onate, E., Dvorkin, E.N. (eds.), Computational Mechanics. New Trends and Applications, *Proc. of Fourth World Congress on Computat.Mechanics*, June 29-July 2, 1998, Buenos Aires, CIMNE (Barcelona), 1998.
5. Plažek, J., Banaś, K., Kitowski, J., and Boryczko, K., Exploiting two-level parallelism in FEM applications, in: B. Hertzberger and P. Slood, (eds.), *Proc.HPCN'97*, April 28-30, 1997, Vienna, Lecture Notes in Computer Science, Vol. 1225 (Springer, Berlin, 1997) 272-281.
6. P. LeTallec, Domain decomposition method in computational mechanics, in: J.T. Oden, ed. (North Holland, Amsterdam, 1994).
7. Plažek, J., Banaś, K., and Kitowski, J., Finite element message-passing/DSM simulation algorithm for parallel computers, in: P. Slood, M. Bubak and B. Hertzberger, (eds.), *Proc.HPCN'98*, April 21-23, 1998, Amsterdam, Lecture Notes in Computer Science, Vol. 1401 (Springer, Berlin, 1998) 878-880.
8. Woodward, P., and Colella, P., The numerical simulation of two dimensional fluid flow with strong shocks, *Journal of Computational Physics*, **54** (1984) 115-173.

A Scalable Parallel Gauss-Seidel and Jacobi Solver for Animal Genetics

Martin Larsen¹ and Per Madsen²

¹ UNI•C, Danish Computing Centre for Research and Education
DTU Building 304, DK-2800 Lyngby, Denmark

² Danish Institute for Agricultural Sciences (DIAS), Research Centre Foulum
P.O. Box 50, DK-8830 Tjele, Denmark

Abstract. A parallel solver based on data parallelism on distributed memory architectures is described. The familiar Gauss-Seidel and Jacobi iterations are used in an "iteration on data" scheme. Minimalization of interprocessor communication is obtained by a split of equations in two sets: Local and Global. Only members of the global set receive contributions from data on more than one processor, hence only global equations require communication during iterations for their solution. Linear and in some models even super linear scalability is obtained in a wide range of model sizes and numbers of parallel processors. The largest example tested in this study is a 3-trait calculation with 30.1 mio. equations.

1 Introduction

Selection of animals in a breeding program is based on breeding values. The true breeding values of animals are unknown. However Estimated Breeding Values (EBVs) can be obtained by solving linear systems of equations called Mixed Model Equations (MMEs) [1]. The MMEs follow from application of BLUP (Best Linear Unbiased Prediction) using a statistical and a genetic model. A best fit of data is done by the statistical model, and animal relationships are described by the genetic model. If all animals in the population are included, the genetic model is called an Animal Model.

The MME for an animal model is sparse. The size of the system is the sum of number of levels, taken over all factors in the model. Because relationships are traced back to a base population, the animal effect can involve a large number of individuals spread over several generations. This can make the MME system very large.

Breeding programs often includes several possibly correlated traits. In order to produce EBVs consistent with the selection criteria multi-trait calculations are needed. Assuming the same model for all traits, the size increases in proportion to the number of traits. The use of more complex statistical models (i.e. including more factors) also increase the dimension of the MME.

In an actual case in Danish Dairy Cattle breeding, the number of equations per trait is approx. 10^7 . Simultaneous estimation for 3 milk production traits, leads to a system with approx. $3 \cdot 10^7$ equations. Although the matrix of the

MME for an animal model is very sparse, it is often too large to build and store explicitly. In the 10 mio. equation case just mentioned, a half-stored sparse representation for the matrix requires approx. 500 GB, while the data is less than 2 GB. Explicit construction of the matrix can be avoided by "iteration on data" as described by Schaeffer and Kennedy [5].

While the size and complexity of MMEs tend to increase, there is also a requirement of more frequent updates of EBVs. At DIAS and UNI•C a joint project aiming at parallelization of an iterative solver for MMEs has taken place from 1995 to 1999. The parallelized solver is based on the familiar Gauss-Seidel (GS) and Jacobi iterations and "iteration on data". Development and test is on a distributed memory architecture and use is made of either PVM or MPI for message passing.

Several different parallel implementations has been developed and tested during the parallelization project [3] [4]. They mainly differ in the amount of communication between processors during iterations.

In this paper, a short description of the linear model in use is given. Then we outline the serial solving strategy. This is followed by a description of the most recent parallel implementation together with test results for parallel speedup.

2 Linear Model

The statistical model describes how the factors (environmental as well as genetic) affect the t measured traits. Suppose β_1 and β_2 are vectors of fixed effects, \mathbf{u}_i , $i=1, 2, \dots, r$, are vectors of random effects for the i^{th} random factor other than the animal factor, \mathbf{a} is a vector of random additive genetic effects, and \mathbf{e} is a vector of random residuals. In a general multivariate linear mixed model the vector \mathbf{y} of observations can be written:

$$\mathbf{y} = \mathbf{X}_1\beta_1 + \mathbf{X}_2\beta_2 + \sum_{i=1}^r \mathbf{Z}_i\mathbf{u}_i + \mathbf{Z}_a\mathbf{a} + \mathbf{e} \quad (1)$$

The structured design matrices \mathbf{X}_1 , \mathbf{X}_2 , \mathbf{Z}_i , $i=1, 2, \dots, r$ and \mathbf{Z}_a are known. Assumptions on expectation values and variance-covariance components are:

$$\begin{aligned} E[\mathbf{u}_i] &= 0 \quad (\forall i), & E[\mathbf{a}] &= 0, & E[\mathbf{e}] &= 0, \\ \text{Var}[\mathbf{e}] &= \mathbf{R} = \mathbf{R}_0 \otimes \mathbf{I}, \\ \text{Var}[\mathbf{u}_i] &= \mathbf{G}_i = \mathbf{G}_{0_i} \otimes \mathbf{I}_i \quad (\forall i), & \text{Var}[\mathbf{a}] &= \mathbf{G}_a = \mathbf{G}_{0_a} \otimes \mathbf{A}, \\ \text{Cov}[\mathbf{u}_i, \mathbf{u}'_j] &= 0 \quad (i \neq j), & \text{Cov}[\mathbf{a}, \mathbf{u}'_i] &= 0 \quad (\forall i), \\ \text{Cov}[\mathbf{u}_i, \mathbf{e}'] &= 0 \quad (\forall i), & \text{Cov}[\mathbf{a}, \mathbf{e}'] &= 0 \end{aligned}$$

where \mathbf{G}_{0_i} are (co)variance matrices for the traits influenced by the i^{th} of the r random effects other than animal, \mathbf{G}_{0_a} is the additive genetic (co)variance matrix and \mathbf{R}_0 is the residual (co)variance matrix for the t traits. \mathbf{A} is the numerator relationship matrix among the animals. Let

$$\mathbf{G} = \bigoplus_{i=1}^r \mathbf{G}_i, \quad \mathbf{Z}' = \begin{bmatrix} \mathbf{Z}'_1; \mathbf{Z}'_2; \dots; \mathbf{Z}'_r \end{bmatrix}, \quad \mathbf{u}' = \begin{bmatrix} \mathbf{u}'_1; \mathbf{u}'_2; \dots; \mathbf{u}'_r \end{bmatrix}$$

where \oplus is the direct sum operator. BLUP estimates of fixed effects $\hat{\beta}_1$ and $\hat{\beta}_2$ and predictions of random effects $\hat{\mathbf{u}}$ and $\hat{\mathbf{a}}$ are found as solutions to the MME:

$$\begin{bmatrix} \mathbf{X}'_1\mathbf{R}^{-1}\mathbf{X}_1 & \mathbf{X}'_1\mathbf{R}^{-1}\mathbf{X}_2 & \mathbf{X}'_1\mathbf{R}^{-1}\mathbf{Z} & \mathbf{X}'_1\mathbf{R}^{-1}\mathbf{Z}_a \\ \mathbf{X}'_2\mathbf{R}^{-1}\mathbf{X}_1 & \mathbf{X}'_2\mathbf{R}^{-1}\mathbf{X}_2 & \mathbf{X}'_2\mathbf{R}^{-1}\mathbf{Z} & \mathbf{X}'_2\mathbf{R}^{-1}\mathbf{Z}_a \\ \mathbf{Z}'\mathbf{R}^{-1}\mathbf{X}_1 & \mathbf{Z}'\mathbf{R}^{-1}\mathbf{X}_2 & \mathbf{Z}'\mathbf{R}^{-1}\mathbf{Z} + \mathbf{G}^{-1} & \mathbf{Z}'\mathbf{R}^{-1}\mathbf{Z}_a \\ \mathbf{Z}'_a\mathbf{R}^{-1}\mathbf{X}_1 & \mathbf{Z}'_a\mathbf{R}^{-1}\mathbf{X}_2 & \mathbf{Z}'_a\mathbf{R}^{-1}\mathbf{Z} & \mathbf{Z}'_a\mathbf{R}^{-1}\mathbf{Z}_a + \mathbf{G}_a^{-1} \end{bmatrix} \begin{bmatrix} \hat{\beta}_1 \\ \hat{\beta}_2 \\ \hat{\mathbf{u}} \\ \hat{\mathbf{a}} \end{bmatrix} = \begin{bmatrix} \mathbf{X}'_1\mathbf{R}^{-1}\mathbf{y} \\ \mathbf{X}'_2\mathbf{R}^{-1}\mathbf{y} \\ \mathbf{Z}'\mathbf{R}^{-1}\mathbf{y} \\ \mathbf{Z}'_a\mathbf{R}^{-1}\mathbf{y} \end{bmatrix} \quad (2)$$

3 Gauss-Seidel and Jacobi Iteration

Write (2) as $\mathbf{C}\mathbf{x} = \mathbf{b}$, and decompose the matrix \mathbf{C} as $\mathbf{C} = \mathbf{L} + \mathbf{D} + \mathbf{U}$ where \mathbf{L} is lower and \mathbf{U} is upper triangular and \mathbf{D} is (block) diagonal. The first order Jacobi (3) and Gauss-Seidel (4) methods are:

$$\mathbf{D}\mathbf{x}^{(k+1)} = -(\mathbf{L} + \mathbf{U})\mathbf{x}^{(k)} + \mathbf{b} = \mathbf{crhs} \Rightarrow \mathbf{x}^{(k+1)} = \mathbf{D}^{-1}\mathbf{crhs} \quad (3)$$

$$\mathbf{x}^{(k+1)} = -\mathbf{D}^{-1}(\mathbf{L}\mathbf{x}^{(k+1)} + \mathbf{U}\mathbf{x}^{(k)}) + \mathbf{D}^{-1}\mathbf{b} = \mathbf{D}^{-1}\mathbf{crhs} \quad (4)$$

where $\mathbf{x}^{(k+1)}$ denotes the $(k+1)^{\text{th}}$ iterate to the solution. \mathbf{crhs} means the corrected right hand side. Rate of convergence in the Jacobi method is often improved by means of a second order Jacobi method [6], i.e. by updating $\mathbf{x}^{(k+1)}$ using a relaxation factor h :

$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k+1)} + h(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}) \quad (5)$$

4 Serial Solving Strategies

"Iteration on data" goes as follows: The first fixed factor (β_1) is chosen as the one containing the largest number of levels. In a preprocessing step, the data records are sorted on the level codes of the first fixed factor, and equation numbers are assigned to all factors affecting the records.

The diagonal blocks $\mathbf{X}'_1\mathbf{R}^{-1}\mathbf{X}_1$ and $\mathbf{Z}'\mathbf{R}^{-1}\mathbf{Z} + \mathbf{G}^{-1}$ are block diagonal on a lower level, with squared blocks of dimensions equal to the number of traits that are affected by the respective fixed or random factors. The diagonal block $\mathbf{X}'_2\mathbf{R}^{-1}\mathbf{X}_2$ for the remaining fixed factors (β_2) is relatively dense but in most applications of limited size. A LU-factorization of this block is stored. The diagonal block for the animal factor $\mathbf{Z}'_a\mathbf{R}^{-1}\mathbf{Z}_a + \mathbf{G}_a^{-1}$ is sparse, but has a complex structure coming from \mathbf{G}_a^{-1} . The matrix $\mathbf{G}_a^{-1} = \mathbf{G}_{0a}^{-1} \otimes \mathbf{A}^{-1}$ is easy to calculate, because \mathbf{G}_{0a} is only of dimension p_a , and \mathbf{A}^{-1} can be formed from pedigree information by simple rules due to Henderson [2]. The non-diagonal blocks shown in (2) are sparse, but can in principle contain nonzeros anywhere. Also, the small diagonal blocks contained in $\mathbf{Z}'\mathbf{R}^{-1}\mathbf{Z} + \mathbf{G}^{-1}$ and $\mathbf{Z}'_a\mathbf{R}^{-1}\mathbf{Z}_a + \mathbf{G}_a^{-1}$ are computed, inverted and stored.

During each round of iteration, the sorted dataset is processed sequentially. Records for one level code for the first fixed factor, is processed as a group.

These records generate one equation in a single trait and t equations in a multi-trait application for the first fixed factor. A new iterate for this equation (t equations) is calculated using the GS formula (4). The same group of records also contributes to matrix elements for the other factors. The products of these contributions and the appropriate components of either $\mathbf{x}^{(k+1)}$ (if available) or $\mathbf{x}^{(k)}$ are absorbed into **crhs**.

When all data are processed, a new iterate for the complete $\hat{\beta}_1$ vector has been produced, and we are left with the equations for $\hat{\beta}_2$, $\hat{\mathbf{u}}$ and $\hat{\mathbf{a}}$ on block diagonal form. New iterates for $\hat{\beta}_2$ are obtained by backsolving based on the LU-factorization. New iterates for $\hat{\mathbf{u}}$ and $\hat{\mathbf{a}}$ are calculated using (3) and (5). Iteration is continued until convergence or a maximum iteration count is reached.

5 Data Distribution

Let r be a data record, \mathcal{R} the set of data records \mathbf{y} in (1) and \mathcal{P} the set of pedigree records for animals included in the relationship matrix \mathbf{A} . Processors are numbered $M = 0, 1, \dots, n-1$ where $M = 0$ is the master. Data are split and distributed in $n-1$ sets $\mathcal{R}(M)$ ($M = 1, 2, \dots, n-1$) fulfilling: $\mathcal{R} = \cup_{M=1}^{n-1} \mathcal{R}(M)$ and $\mathcal{R}(M) \cap \mathcal{R}(M') = \emptyset \forall M, M' : M \neq M'$.

Records contributing to the same equation (block of equations in multi-trait) from the first fixed factor, are mapped to the same slave, i.e. β_1 is split in disjoint sets ($\beta_1(M)$, $M = 1, 2, \dots, n-1$). Let $\mathcal{F}(M)$ be the set of references in **crhs** generated by the records in $\mathcal{R}(M)$, and \mathcal{A} be a vector of counters with length equal to number of equations (blocks) in the MME. $\mathcal{A}(i)$ is incremented by 1 when equation (block) i is referenced for the first time on slave M .

The master processor ($M = 0$) executes the data distribution algorithm as follows: The slave number S is set to 1 and $\mathcal{F}(S)$ is initialized to \emptyset . An r is read sequentially from \mathcal{R} and $\mathcal{F}(S)$ is updated with references generated by r . Then the next r is read and processed similarly etc. When $\mathcal{R}(S)$ is completed, it is sent to slave S . $\mathcal{F}(S)$ is updated with the subset of \mathcal{P} related to $\mathcal{R}(S)$ and sent to slave S . \mathcal{A} is updated from $\mathcal{F}(S)$, $\mathcal{F}(S)$ is reinitialized to \emptyset and S is updated as $S = S+1$. This goes on $\forall r \in \mathcal{R}$. Finally \mathcal{A} is broadcasted to all slaves.

On slave $M = 1, 2, \dots, n-1$, data distribution implies the following processing: The set of records $\mathcal{R}(M)$ is received. As large a proportion of $\mathcal{R}(M)$ as possible is stored in memory, the remaining proportion on disk. Then $\mathcal{F}(M)$ and \mathcal{A} are received. Equations (blocks) on slave M are split in local ($\mathcal{L}(M)$) and global (\mathcal{G}) sets defined as: $\mathcal{L}(M) = \{i \mid \mathcal{A}(i) = 1 \wedge i \in \mathcal{F}(M)\}$ and $\mathcal{G} = \{i \mid \mathcal{A}(i) > 1\}$.

References for $\mathcal{L}(M)$ and \mathcal{G} are transformed to a local vector space on slave M . Only equations (blocks) in \mathcal{G} generates communication in the iterative solver. To simplify this communication, an ordered version of \mathcal{G} is allocated on all slaves.

To parallelize adjustments from \mathbf{G}_a^{-1} during iterations, all slaves extract subsets $\mathcal{P}_L(M)$ of \mathcal{P} corresponding to animals in $\mathcal{L}(M)$. Slave $M = 1$ also extracts a subset \mathcal{P}_G corresponding to animal equations in \mathcal{G} . These subsets fulfills:

$$\mathcal{P} = (\cup_{M=1}^{n-1} \mathcal{P}_L(M)) \cup \mathcal{P}_G ; \mathcal{P}_G \cap \mathcal{P}_L(M) = \mathcal{P}_L(M) \cap \mathcal{P}_L(M') = \emptyset \forall M, M' : M \neq M'.$$

6 Parallel Iteration

Let the local and global subsets of \mathbf{u} and \mathbf{a} on M be $\mathbf{u}(M)$ and $\mathbf{a}(M)$ respectively and $\mathbf{x}' = [\beta'_1; \beta'_2; \mathbf{u}'; \mathbf{a}']$. In pseudocode the parallel iterative solver is thus:

```

 $\mathbf{x}^{(0)} = \mathbf{0}; k = 0; cd = +\infty$ 
do while ( $cd \geq \epsilon \wedge k \leq \text{maxiter}$ )
   $k = k + 1$ 
  if ( $M \geq 1$ ) then
    do  $\forall r \in \mathcal{R}(M)$ 
      build equations for  $\beta_1(M)$ 
      absorb  $\beta_2^{(k-1)}$ ,  $\mathbf{u}(M)^{(k-1)}$  and  $\mathbf{a}(M)^{(k-1)}$  into crhs
      solve for  $\beta_1(M)^{(k)}$ 
      absorb  $\beta_1(M)^{(k)}$  into crhs
    enddo
    process  $\mathcal{P}_L(M)$ 
    if ( $M = 1$ ) process  $\mathcal{P}_G$ 
    send subset of crhs corresponding to  $\mathcal{G}$  to  $M = 0$ 
    receive summed crhs corresponding to  $\mathcal{G}$  from  $M = 0$ 
    solve for  $\mathbf{u}(M)^{(k)}$  and  $\mathbf{a}(M)^{(k)}$ 
    receive  $\beta_2^{(k)}$  from  $M = 0$ 
    calculate local contributions to  $\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_2^2$  and  $\|\mathbf{x}^{(k)}\|_2^2$ 
    recursive doubling of  $\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_2^2$  and  $\|\mathbf{x}^{(k)}\|_2^2$ 
  else
    receive and sum subset of crhs corresponding to  $\mathcal{G}$  from all slaves
    broadcast summed subset of crhs corresponding to  $\mathcal{G}$ 
    backsolve for  $\beta_2^{(k)}$  from LU-decomposition
    broadcast  $\beta_2^{(k)}$ 
    calculate local contributions to  $\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_2^2$  and  $\|\mathbf{x}^{(k)}\|_2^2$ 
    recursive doubling of  $\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_2^2$  and  $\|\mathbf{x}^{(k)}\|_2^2$ 
  endif
   $cd = \|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_2 / \|\mathbf{x}^{(k)}\|_2$ 
enddo
collect  $\mathbf{x}^{(k)}$  on  $M = 0$ 

```

7 Test Scenarios

Tests has been made on the following four data sets:

1. Protein yield for all Red Danish dairy cattle (RD).
2. Protein, fat and milk yield for all Red Danish dairy cattle (RD3).
3. Protein yield for all dairy cattle in Denmark (AC).
4. Protein, fat and milk yield for all dairy cattle in Denmark (AC3).

The dimensions of the MME to solve were 1.2, 4.3, 9.5 and 30.1 mio. for RD, RD3, AC and AC3 respectively.

The computer environment was an IBM RS/6000 SP with 80 processors (48 with 512 MB and 32 with 256 MB memory) for parallel computation. The tests was performed on processors with 512 MB memory.

8 Test Results

The execution time per round of iteration is shown in Figure 1. Systems of this kind require between 1000 and 3000 rounds of iteration for convergence (dependent on model). PVM was used for message passing.

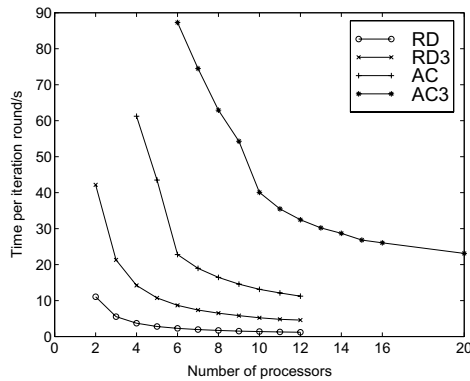


Fig. 1. Real time per round of iteration depending on number of processors for solving RD, RD3, AC and AC3 models (containing 1.2, 4.3, 9.5 and 30.1 mio. equations respectively).

A measure of parallel speedup is relative speed, defined as reciprocal elapse time normalized to a definite value on a definite number of processors.

Figure 2 show the speedup for all four test models. Relative speed has been normalized so that the speed is set to 6 when executed on 6 processors. This is the smallest number of processors on which all test models can be executed due to memory requirement.

For RD and RD3 close to linear speedup is obtained. From 2 to 8 processors super linear scalability is obtained. Going from 2 to 4 processors results in a 3-fold increase in speed. The AC model shows super scalability when going from 4 to 6 processors because only a part of data can be in memory on less than 6 processors. Similar patterns for speedup is seen for AC3, but here the super scalability continues until 12 processors. From 12 to 16 processors the speedup is linear and a further increase in number of processors results in less than linear scalability. Recently a MPI version has been developed. It has only been tested in the AC3 example, where it is approx. 15 % faster than the PVM version.

As the number of processors increase, the data (\mathcal{R}) is split in an increasing number of subsets ($\mathcal{R}(M)$). Thus the number of global equations ($\#\mathcal{G}$) increase.

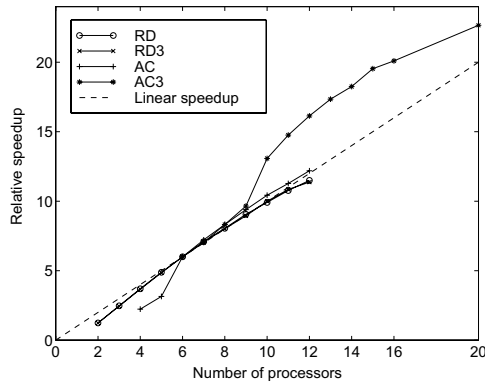


Fig. 2. Relative speed depending on number of processors for solving RD, RD3, AC and AC3 models. Speed is normalized to 6 on 6 processors.

This can explain the tendency of the speedup curves of Figure 2 to fade out at sufficiently many processors. It leads to more communication and computation because the global equations are solved on all slave processors. From 6 to 20 processors in the AC3 example, $\#\mathcal{G}$ increase from 1.2 to 1.7 mio. The total number of equations solved on all processors increase from 33.5 to 58.9 mio. The number of equations per slave processor and the proportion of these which is global is shown for the AC3 example in Figure 3.

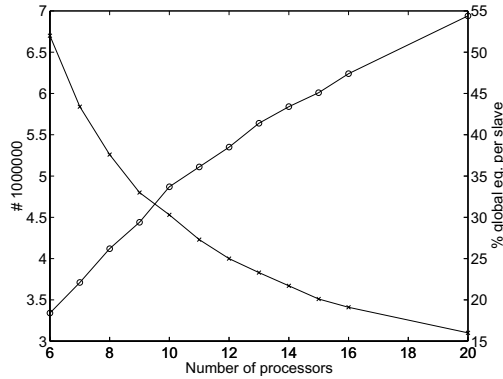


Fig. 3. Number of equations solved per slave processor (x) and the proportion (in percent) of global equations (o) depending on number of processors. AC3 example.

9 Further Improvements

The following improvements are under consideration:

1. Reduce the number of animals with data records on more than one processor by making use of relationship information during the data distribution phase. This can minimize the increase in $\#\mathcal{G}$ with increase in number of processors.
2. Reduce the number of equations solved on the slave processors by only solving the local equations and the subset of \mathcal{G} linked to $\mathcal{R}(\mathbf{M})$. This will limit the increase in the total number of equations solved with increasing number of processors, but will also require mapping vectors for formation of the global part of **crhs**.
3. In the present implementation, the workload on the master processor is low during iterations. The data distribution and solver code can be changed so the master perform iteration on data like the slaves but preferably on a slightly smaller set of records in order to allow for tasks specific for the master.

10 Conclusion

The parallel solver implementation based on distinction between local and global equations show good scalability both over number of processors and over dimension of the MME equation system. This opens for solving larger systems or for solving problems in shorter time.

Acknowledgments

This work was supported with computing resources from the Danish Natural Science Research Council. From 1997 to 1999 the project has received support from EU under the title Continuous Estimation of Breeding Values Using Supercomputers (CEBUS, Esprit Project no. 24721).

References

1. Henderson C.R., 1973. *Sire evaluation and genetic trends*. In Proc. of the Animal Breeding and Genetics Symposium in Honor of Dr. D.L.Lush. ASAS and ADSA, Champaign, Ill.
2. Henderson C.R., 1976. *A simple method for computing the inverse of a numerator relationship matrix used in prediction of breeding values*. Biom. 32:69-83
3. Larsen M. and P. Madsen, 1999. *The CEBUS project: History and overview*. Proc. of the CCB'99 International Workshop on High Performance Computing and New statistical Methods in Dairy Cattle Breeding. Interbull Bulletin. In print.
4. Madsen P. and M. Larsen, 1999. *Attacking the problem of scalability in parallel Gauss-Seidel and Jacobi solvers for mixed model equations*. Proc. of the CCB'99 International Workshop on High Performance Computing and New statistical Methods in Dairy Cattle Breeding. Interbull Bulletin. In print.
5. Schaeffer L.R. and B.W. Kennedy., 1986. *Computing Solutions to Mixed Model Equations*. Proc. 3.rd. World Congr. on Genetics Applied to Livest. Prod. 12, 382-393. Nebraska.
6. Misztal, I. and D. Gianola, 1987. *Indirect Solution of Mixed Model Equations*. J. Dairy Sci. 70, 716-723.

Parallel Approaches to a Numerically-Intensive Application Using PVM

R. Baraglia¹, R. Ferrini¹, D. Laforenza¹, and A. Laganà²

¹ CNUCE - Institute of the Italian National Research Council
Via S. Maria, 36 - I-56100 Pisa, Italy

Tel. +39-050-593111 - Fax +39-050-904052

{R.Baraglia,R.Ferrini,D.Laforenza}@cnuce.cnr.it

² Department of Chemistry, University of Perugia

Via Elce di Sotto, 8 - I-06123 Perugia, Italy

Tel. +39-075-5855515 - Fax +39-075-5855606

lag@unipg.it

Abstract. This paper proposes some schemes for the parallelization of a quantum reactive scattering code that integrates a two dimensional Schrödinger equation. The parallelization is based on both task farm and pipeline models whose processes communicate via the message-passing paradigm. The various versions of the code are implemented using the Parallel Virtual Machine (PVM) communication library. Benchmarks of the two proposed parallel models are performed on a CRAY T3E system and the measured performances are then discussed.

1 Introduction

The calculation of efficiency parameters, such as cross sections, for gas phase elementary reactions, is a key step in the realistic simulation of several environmental and technological problems. This is a many body problem that nowadays can be solved using either classical or quantum mechanics [1].

The most popular way of solving this problem is to integrate in time the classical equations describing the motion of the atoms of the molecular system, under the assumption that they behave as structureless mass points. On the basis of this assumption, relates computer codes can deal independently with a large number of collisional events and, as a result, can be efficiently implemented on parallel machines [2]. However, classical mechanics is by definition insensitive to the quantum effects (tunneling, resonances and interferences) that characterize cross sections at some critical values of the collision energy (for example at thresholds). The appropriate treatment of these effects can only be carried out using quantum methods. In this paper, we consider the quantum Reactive Infinite Order Sudden (RIOS) method [3,4], which reduces the dimensionality of the bimolecular reaction problem to 2, by decoupling both orbital and rotational motions. It then leads to the integration of a set of fixed collision angle differential equations.

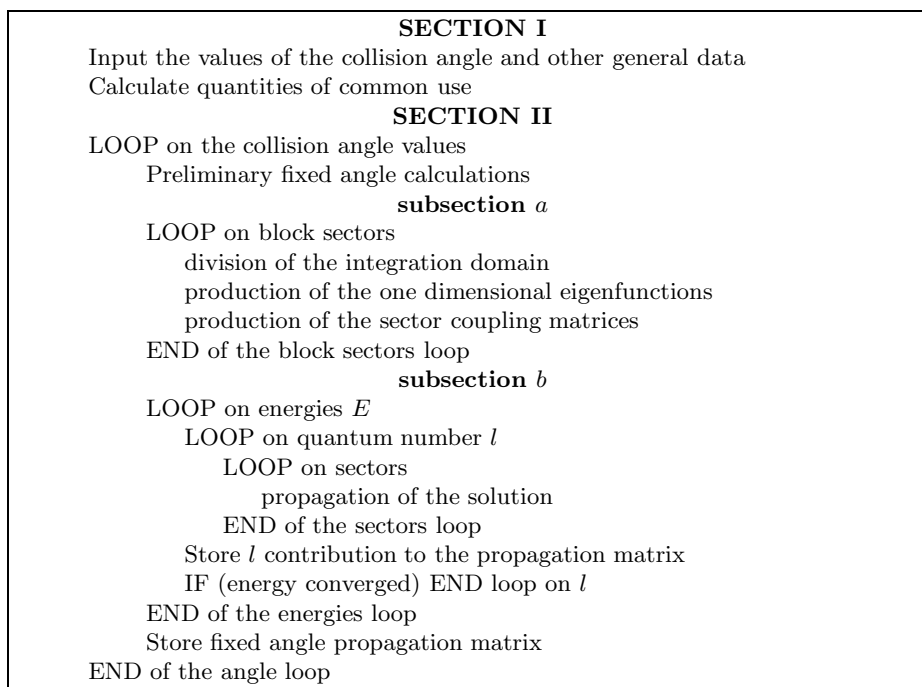


Fig. 1. Scheme of the RIOS algorithm.

To integrate these equations a computer code using a propagation technique has been assembled. Its flow scheme is given in Fig. 1. As can be seen, the main body of the program (*SECTION II*) is a loop over the values of the collision angle which is divided into two subsections. In subsection *a* the fixed collision angle reaction channel is divided into many small sectors, and the related basic functions, overlaps and coupling matrices are calculated. In subsection *b* the solution of the fixed angle coupled differential equations is propagated from reactants to products [3].

In this paper we first discuss the level of complexity of the parallel models adopted to implement the code on massively parallel architectures. Then performances measured when running the code on a CRAY T3E are analyzed and explained in terms of the characteristics of the models.

2 Parallelization Strategies

In order to adopt an efficient parallelization strategy some aspects of the problem decomposition, which are typical of quantum methods, have been analyzed in detail [5,6]. As already mentioned, Fig. 1 shows that the main body of the computation is performed in *SECTION II*. This section consists of a loop over

the collision angle values, which is divided into two subsections with different memory and cpu time requirements.

In subsection *a* the reaction channel is divided into several hundred one-dimensional cuts (each cut consisting of a grid of at least a thousand points) on which eigenvalues and eigenfunctions of the bound coordinate are calculated. From these values the matrices that map the solution at the border of adjacent sectors and couple the various differential equations in the reaction coordinate are computed and stored. In subsection *b* the solution matrix (i.e., the matrix of the ratio between the function and its derivative) is initialized and then propagated through the sectors for a given pair of E and l values (an $E_k l_m$ pair) using the mapping and coupling matrices produced in subsection *a*.

To design a suitable parallel strategy, cpu and memory requirements, as well as order dependencies of the various parts of the code, are examined in order to find out the amount of computational work to be assigned to individual processes (granularity). This information is necessary to organize the code concurrently.

When designing a parallelization strategy, a commonly used procedure is to refer to well established prototypes (parallel models) such as the task farm and the pipeline ones [7].

2.1 Task Farm Model

Let us first consider fixed collision angle calculations, since each iteration of the loop over the value of the collision angle can be executed independently. In the task farm model [8] a static data decomposition can be performed to parallelize subsection *a*. To this end, sectors can be grouped into blocks of approximately the same size and each block is assigned to a different processor. The number of sector blocks generated in this way depends on the number of processors available on the machine used. At the end of subsection *a*, the coupling matrix is broadcast and stored in the memory of all the nodes. This means that the computation of subsection *b* can only start after the last coupling matrix contribution has been calculated and broadcast.

To parallelize subsection *b* a dynamic assignment of the work is more convenient. In this case, a node has to act as the master of the farm and take charge of assigning a packet of work units of predetermined granularity to the slave nodes. The work unit is a fixed angle propagation of the solution from reactants to products for a given $E_k l_m$ pair. Work units are executed sequentially by the slaves. The convergence of the results with l is checked before assigning further calculations for different l values propagations at constant E .

A close analysis of the structure of the task farm model highlights several potential sources of inefficiency, such as the communication times associated with the transfer of the coupling matrix and of the solution matrix, the synchronization times associated with the alignment of the execution of subsections *a* and *b*, the unproductive times spent on propagating the solution for $E_k l_m$ pairs before realizing that convergence with l has been reached, and the waiting times associated with final load imbalancing.

To improve the efficiency of the task farm parallel model, the nodes of the parallel machine can be clustered into groups (clusters) of approximately the same size, and parallelism can be extended to the next (upper) level that related to the iteration on the collision angle. The master process is in charge of coordinating both levels of calculation and synchronization for the two subsections. The final load imbalance was minimized by dynamically resizing the clusters at run time, despite the fact that this introduces an overhead associated with the time needed to supply the new coupling matrix to the nodes moved to a different cluster.

2.2 Pipeline Model

The pipeline model pushes the parallelism to a lower level (to a finer granularity) by partitioning into blocks of sectors not only the calculation of the coupling matrix but also the propagation of the solution [9]. Each node thus only needs to have in memory the portion of the coupling matrix related to the sectors for which it has carried out the calculation of the eigenvalues and eigenfunctions. Consequently, while subsection *a* has the same structure as in the task farm model (without, however, performing the broadcast of the calculated portion of the coupling matrix), the parallel structure of subsection *b* has been radically changed. Each node, in fact, is no longer in charge of carrying out the whole propagation (from reactants to products) for an $E_k l_m$ pair. Moreover, individual nodes only propagate the solution through the block of sectors for which they have calculated the coupling matrix in subsection *a*. The solution is then passed to the next node, which propagates the solution through the next block of sectors while the previous node starts a new propagation for another $E_k l_m$ pair. Convergence with l is checked by the last processor in the pipe. When convergence is reached, the last processor issues a message to the remaining nodes asking them to terminate currently active propagations for higher values of l at the same energy.

The inefficiency sources of the pipeline model are the startup time (the time needed to fill up the pipe), the synchronization time (the time needed to transfer and/or receive the results from one sector block to the next one), the unused time (the time spent carrying out the propagation for those $E_k l_m$ pairs which will not contribute to the final result because convergence has already been reached), and the final load imbalance time. One way of reducing the largest inefficiency times, is to organize the sectors into blocks made up of a different number of sectors (a sector distribution which linearly increases with the node number was found to be an optimum choice [9]). Also, to minimize the unused calculation time, we took advantage of the fact that contributions to the cross section are constantly monitored and a *preconvergence state* can be defined. The preconvergence state is a state in which the reactive probability is smaller than a given threshold value. Consequently, when the last node, which carries out the final propagation and the convergence check, finds that fixed l contributions to the cross section (at a given E value) have become smaller than the threshold value, a message is

issued to the first node to let it start the propagation for the first l value of the next energy.

As for the task farm model, the nodes can be grouped into clusters to carry out fixed angle calculations. The main advantage of partitioning the machine into clusters is that the startup time, the synchronization time, and the unused time associated with shorter pipelines, are smaller. The clustering entails managing the distribution among the clusters of the fixed angle calculations by a master, as in a task farm model. The optimized parallel model thus turns out to be a hybrid task farm of pipelines.

3 Parallel Implementation on Cray T3E

For the implementation of the task farm model on a Cray T3E, we chose a *medium* grain (one energy to each processor at a time). The communications among the nodes were developed by using the message-passing paradigm, and implemented with the Parallel Virtual Machine (PVM) [10] communication library. Since the Cray T3E uses the SPMD programming scheme, one node has to be used to run the master process. In this way a processing element is not available for execution. This model often leads to clusters with different dimensions. At run time this cluster configuration can lead to large final load imbalance times.

The implementation of the pipeline model implies that one node of the Cray T3E is used as a master node, while the remaining nodes are partitioned into clusters. Each cluster carries out a fixed angle calculation by using a pipeline mechanism. As in the task farm model, the communication among the nodes was implemented by using the facilities of the PVM communication library.

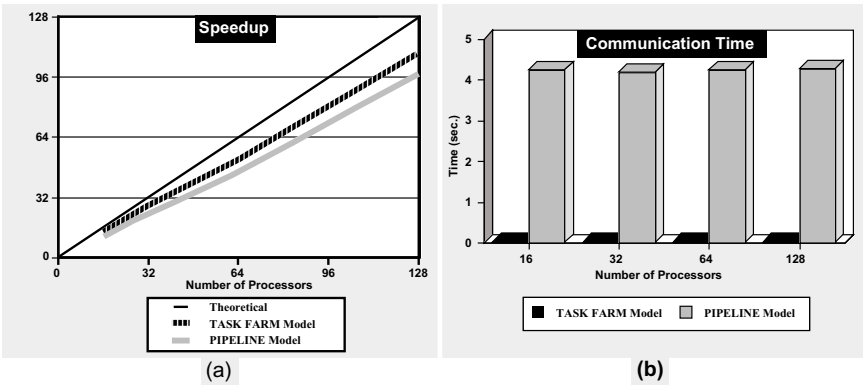


Fig. 2. Speedups (a) and communication times (b) of the task farm and pipeline model on a Cray T3E1200 architecture.

3.1 Performance Evaluation

The Task Farm and Pipeline versions of RIOS were run on a Cray T3E1200 that have 128 nodes with a local memory of 256 Mbytes, and using the following parameters: 64 angles, 80 (fairly high) energies, 400 sectors, 80 quantum numbers, and 15 vibrational states. To scale up the problem, runs with 16, 32, 64 and 128 processors to compute 8, 16, 32 and 64 collision angles, respectively, were made. In the four tests the nodes of the machine were grouped into 1, 3, 7 and 15 clusters of eight nodes, and one cluster of seven nodes. In all the runs the main inefficiency factors (i.e. startup time, synchronization time and final load imbalance time) of the task farm and pipeline models were monitored. Fig. 2 (a) gives the speedup values of the two versions when varying the number of processors. The speedup values indicate that the performance of the two models was satisfactory. However, as the figure shows, the task farm version leads to better performances than the pipeline one.

Communication, synchronization and load imbalance times were normalized with respect to the number of computed energies.

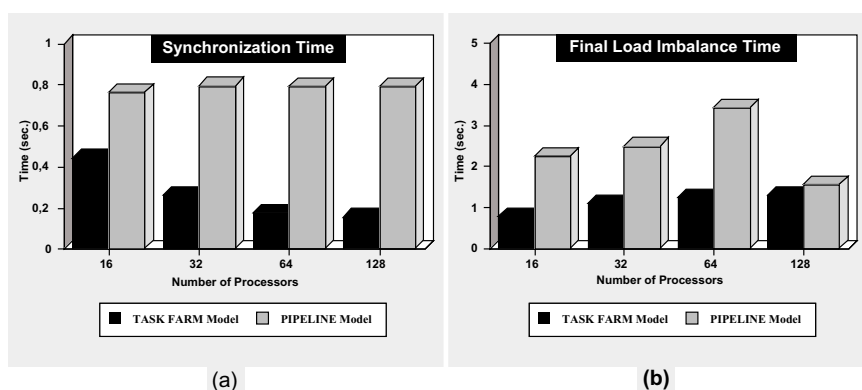


Fig. 3. Synchronization (a) and load imbalance times (b) of the task farm and pipeline model on a Cray T3E1200 architecture.

Communication times include the time spent while transferring data and waiting to read the buffer message. For the task farm model, the largest fraction of communication time measured for subsection *a* is associated with the broadcast of the various blocks of the coupling matrix from the computing node to all the other nodes, while the largest fraction measured for subsection *b* is associated with the transfer of the coupling matrix to nodes moved from an inactive (stopped) cluster to an active cluster. In the pipeline model, the communication time includes the startup times that are associated with the various fixed energy calculations.

As shown in Fig. 2 (b) the communication time of the pipeline model is larger than that of the task farm model.

Fig. 3 (a) reports the single energy synchronization time, measured for the pipeline and the task farm models when varying the number of processors. The synchronization times of the pipeline model are longer than those of the task farm model.

The final load imbalance of the two parallel versions is shown in Fig. 3 (b). In the task farm model, the possibility of resizing the cluster at run time allows the idle time of the slave nodes to be reduced, and this leads to a reduction in the final load imbalance. This confirms that the task farm model should be chosen to make when parallelizing the RIOS code, except when the coupling matrix becomes so large that it cannot be fitted into the node memory (as may occur with very large molecules).

4 Conclusions

Quantum theoretical chemists are increasingly being attracted by the possibility of exploiting the high performance features of modern parallel architectures. The computing power offered by these machines, along with the advantages obtained by grouping them into a heterogeneous cluster, seem to be the ideal ground for a fast development of complex computational procedures that can provide a realistic modeling of chemical processes.

In this paper, we have examined some elementary and composite models that are suitable for parallelizing time-independent quantum reactive scattering codes. The inefficiency factors of various simple and composite parallel models have been analyzed for runs of the RIOS program performed on the CRAY T3E. Due to the non unique correspondence between the parameters of the physical problem and the parameters of the algorithm, the response of the systems may vary significantly as a function of the physical parameters depending on the particular machine being used. For this reason we found it useful to implement different parallel structures and to make any choices in relation to the available resources.

Acknowledgment

The authors would like to thank the CNUCE-Institute of the Italian National Research Council and the CINECA supercomputing center for allowing us to use their parallel systems. Financial support from Italian Space Agency and from NATO (grant N. CRG 920051) is acknowledged.

References

1. *Non Equilibrium Processes in Partially Ionized Gases*. M. Capitelli and J.N. Bardsley Eds. Plenum, New York, 1990.
2. A. Laganà, E. Garcia, O. Gervasi, R. Baraglia, D. Laforenza, and R. Perego. *Theor. Chim. Acta*, 79:323, 1991.
3. A. Laganà, E. Garcia, and O. Gervasi. *J. Chem. Phys.*, 89:7238–7241, 1988.
4. M. Baer, E. Garcia, A. Laganà, and O. Gervasi. *Chem. Phys. Letters*, 158:362, 1989.
5. G. Fox, R. D. Williams, and P. C. Messina. *Parallel Computing Works!*. Morgan Kaufmann Publishers, Inc, 1994.
6. R. Baraglia, D. Laforenza, and A. Laganà. Parallelization Strategies for a Reduced Dimensionality Calculation of Quantum Reactive Scattering Cross section on a Hypercube Machine. *Lecture Notes in Computer Science*, 919:554–561, 1995.
7. A.J.G. Hey. Experiment in MIMD Parallelism. *Proc. of International Conference PARLE'89*, 28–42, 1989.
8. R. Baraglia, R. Ferrini, D. Laforenza, and A. Laganà. On the Optimization of a Task Farm Model for the Parallel Integration of a Two-Dimensional Schrödinger Equations on Distributed Memory Architectures. *ICA3PP, IEEE Third International Conference on Algorithms and Architectures for Parallel Processing*, 543–556, December 1997.
9. R. Baraglia, R. Ferrini, D. Laforenza, and A. Laganà. On the Optimization of a Pipeline Model to Integrate Reduced Dimensionality Schrödinger Equations for Distributed Memory Architectures. *The International Journal of Supercomputer Applications and High Performance Computing*, Vol. 13, no. 1, Spring 1999, pp. 49–62.
10. A. L. Beguelin, J. J. Dongarra, G. A. Geist, W. Jiang, R. Mancheck, and V. S. Sunderam. PVM: Parallel Virtual Machine: A users' Guide and Tutorial for Networked Parallel Computing. *The MIT Press*, 1994.

Solving the Inverse Toeplitz Eigenproblem Using ScaLAPACK and MPI*

J.M. Badía¹ and A.M. Vidal²

¹ Dpto. Informática., Univ Jaume I. 12071, Castellón, Spain.
badia@inf.uji.es

² Dpto. Sistemas Informáticos y Computación. Univ. Politécnica de Valencia.
46071, Valencia, Spain.
avidal@dsic.upv.es

Abstract. In this paper we present a parallel algorithm for solving the inverse eigenvalue problem for real symmetric Toeplitz matrices. The algorithm is implemented using the parallel library ScaLAPACK, based on the communication model MPI. Experimental results are reported on a SGI Power Challenge and on a cluster of PC's connected by a Myrinet network. The results show that the algorithm is portable and scalable.

1 Introduction

The sequential and parallel linear algebra libraries LAPACK and ScaLAPACK, respectively, have become basic tools to tackle any kind of numerical problems in Physics and in Engineering.

ScaLAPACK [1] is a library of linear algebra routines for message-passing distributed memory computers. As in LAPACK, ScaLAPACK routines are based in block-oriented algorithms to minimize the data movement between different levels of the memory hierarchy. The main ScaLAPACK components are BLACS and PBLAS. BLACS is a message-passing library, which includes a set of routines to perform the communications that appear in parallel linear algebra algorithms. There are implementations of the BLACS library using MPI [2], PVM and other message-passing environments. PBLAS includes parallel versions of Level 1, 2 and 3 BLAS routines implemented for distributed memory using the message-passing programming model.

In this work, we analyze the parallel solution of a complex linear algebra problem using the ScaLAPACK library and the message-passing environment MPI. The problem is the inverse eigenvalue problem for Real Symmetric Toeplitz (RST) matrices.

* This paper was partially supported by the project CICYT TIC96-1062-C03: "Parallel Algorithms for the computation of the eigenvalues of sparse and structured matrices".

Let $t = [t_0, t_1, \dots, t_{n-1}]$ where t_i are real numbers. We say $T(t)$ is a real symmetric Toeplitz matrix, generated by t , if

$$T(t) = \left(t_{i-j} \right)_{i,j=1}^n.$$

This type of matrices appears in relation with the solution of several problems in different areas of Physics and Engineering. Moreover, the problem is specially appropriate for a parallel implementation due to its big computational cost.

In this paper we present a parallel implementation for solving the previous problem. Moreover, we report the experimental performance on two machines, a SGI Power Challenge multiprocessor and a cluster of PC's linked by a Myrinet network. In the last case, we compare the experimental results obtained with two different versions of the MPI environment. These versions have been specifically implemented for this gigabit network. A more detailed description of the method and of the experimental results can be found in [3].

The rest of the paper has the following structure: in section 2 we briefly describe the problem to be solved; in section 3 we present the sequential method utilized; the parallelization problem is studied in section 4, and last, in section 5 we report the obtained experimental results.

2 The Inverse Eigenvalue Problem

Let us denote of the eigenvalues of the matrix $T(t)$ by $\lambda_1(t) \leq \lambda_2(t) \leq \dots \leq \lambda_n(t)$.

We say an eigenvector, $x = [x_1, x_2, \dots, x_n]$, of matrix $T(t)$ is *symmetric* if

$$x_j = x_{n-j+1}, \quad 1 \leq j \leq n$$

and is *skew-symmetric* if

$$x_j = -x_{n-j+1}, \quad 1 \leq j \leq n.$$

Moreover, we will call *even (odd)* eigenvalues to those eigenvalues associated with the symmetric (skew-symmetric).

In [4] it is shown that if $r = \lceil n/2 \rceil$, $s = \lfloor n/2 \rfloor$, and T is a RST matrix of order n , then \Re^n has an orthonormal basis consisting of r symmetric and s skew-symmetric eigenvectors of T .

In addition, there exist a set of properties of Toeplitz matrices which allows us to calculate the odd and even eigenvalues separately. This calculation is performed from T by constructing two matrices with half order each, thus providing a substantial saving in the computational time.

In [5] a method for solving the inverse Toeplitz eigenvalue problem, equivalent to the Newton method, is proposed. This algorithm is improved in [6] by adequately exploiting the previous properties of Toeplitz matrices.

Now, we state the problem to be solved in similar terms to those in [6]:

Given n real numbers $\mu_1 \leq \mu_2 \leq \dots \leq \mu_r$ and $v_1 \leq v_2 \leq \dots \leq v_s$, find an n -vector t such that

$$\mu_i(t) = \mu_i, \quad 1 \leq i \leq r, \quad \text{and} \quad v_i(t) = v_i, \quad 1 \leq i \leq s,$$

where the values $\mu_i(t)$ and $v_i(t)$ are, respectively, the even and odd eigenvalues of the RST matrix $T(t)$.

3 Sequential Algorithm

In this section we briefly describe algorithmically, the sequential method proposed in [6] for solving the inverse eigenvalue problem for a RST matrix.

First, we denote by $p_1(t), \dots, p_r(t)$ the symmetric eigenvectors of $T(t)$, and by $q_1(t), \dots, q_s(t)$ the skew-symmetric eigenvectors. In addition we denote the target spectrum as

$$\Lambda = [\mu_1, \mu_2, \dots, \mu_r, v_1, v_2, \dots, v_s], \quad (1)$$

where the even and odd spectra have been separated and reordered in increasing order.

Let t^0 be an n -vector, and let Λ be the target spectrum such as it is defined in (1). The basic idea of the algorithm is the following: by using t^0 as an starting generator, compute a sequence $t^m, m=1, 2, \dots$, as the solution of the equations

$$\begin{aligned} p_i(t^{m-1})^T T(t^m) p_i(t^{m-1}) &= \mu_i, \quad 1 \leq i \leq r \\ q_j(t^{m-1})^T T(t^m) q_j(t^{m-1}) &= \mu_j, \quad 1 \leq j \leq s. \end{aligned} \quad (2)$$

The solution of these equations can be rewritten as the solution of a linear system of size $n=r+s$.

If we denote by $\Lambda(t)$ the spectrum of $T(t)$ and by Λ the target spectrum, we can express the distance between both spectra as

$$\sigma(t; \Lambda) = \|\Lambda(t) - \Lambda\|_2. \quad (3)$$

We will say that the above method has converged if

$$\sigma(t^m; \Lambda) < \varepsilon_0. \quad (4)$$

for some integer m , where ε_0 defines the desired precision for the result.

We will define a basic iterative algorithm `alg1` with the following specification:

ALGORITHM [D', t, fails] = `alg1`(t_0, n, D, eps_0)

Given an initial generator t_0 of size n , a target spectrum D , and a real value eps_0 which defines the desired precision in (4), `alg1` returns the generator t of a RST matrix, its computed spectrum D' and a boolean value `fails` which indicates if the algorithm has failed to converge (5).

In each iteration of the previous algorithm, the initial task is to construct the matrix of the linear system, which allows us to solve (2). This is performed from the eigenvectors of the RST matrix computed in the previous iteration. Then, the linear system is solved, and its solution provides a new generator for RST matrix. The spectrum of this matrix is computed and the convergence is reached if (4) is verified.

As the Newton method does not converge globally, the algorithm `alg1` does not necessary converge to the solution of the problem. We say that the Algorithm 1 has failed if

$$\sigma(t^k; \Lambda) \geq \sigma(t^{k-1}; \Lambda) \geq \varepsilon_0. \quad (5)$$

In [6] an improvement of `alg1` is proposed. The algorithm is organized in two stages: During the first stage, if the algorithm fails, a new target spectrum such as

$$(1 - \rho)\Lambda + \rho\Lambda(t^{k-1}) \quad (6)$$

is proposed, where ρ is a real value in $(0,1)$. During this first stage the value of ρ is modified until the convergence to the target spectrum is reached linearly, with a less restrictive stopping criterion. From this new point the second stage starts and the algorithm `alg1` is utilized in order to converge to the target spectrum quadratically. This algorithm can be specified as follows:

Algorithm 2. Given an initial generator t_0 of size n , a target spectrum D , real values eps_0 , eps_1 and alfa , associated to the convergence criteria of the linear and quadratic stage, and a real value dro , the following algorithm returns the generator t of a RST matrix, its computed spectrum D' and a boolean value `fails` which indicates if the algorithm failed to converge.

```

ALGORITHM [D', t, fails] = alg2(t0, n, D, eps0, eps1, alfa, dro)
BEGIN
  [D', t, fails] = alg1(t0, n, D, eps0)
  IF fails THEN (* linear stage *)
    [D(t0), P] = compute_spectrum(t0, n)
    ro = 0.1; error = ||D(t0) - D||
    WHILE (error > eps1) AND (ro < 1) DO
      Dmod = (1 - ro) * D + ro * D(tk)
      [D(tk+1), tk+1, fails] = alg1(tk, n, Dmod, alfa*error)
      IF fails THEN
        ro = ro + dro; tk = t0
      ELSE
        tk = tk+1
      ENDIF
      error = ||D(tk) - D||
    ENDWHILE
    IF (ro > 1) THEN
      fails = TRUE
    ELSE (* quadratic stage *)

```



```

[D', t, fails] = alg1(tk, n, D, eps0)
ENDIF
ENDIF
END.

```

In the previous algorithm the subroutine $[D(t), P] = \text{compute_spectrum}(t, n)$ returns the eigenvalues $D(t)$ and eigenvectors P of a matrix $T(t)$ of size $n \times n$. This subroutine exploits the properties mentioned in section 2, halving the cost of computing the spectrum, and maintains separated the even and odd spectra.

3 Parallel Algorithm

In order to parallelize the above method we have used the numerical linear algebra library ScaLAPACK. We try to obtain a portable and scalable algorithm for distributed memory systems. We have also used the message-passing interface MPI to exploit the excellent implementations of this environment. The parallel algorithm is implemented in a mesh of processors using an SPMD model, and the matrices are distributed using a block cyclic scheme.

Algorithm 2 is based on four subroutines. The first subroutine computes the spectrum of a RST matrix using the spectra of two smaller symmetric matrices. To perform this task we have used the ScaLAPACK routine `PDSYEV`. As the eigenvectors of the two matrices are independently distributed in the mesh, we have redistributed them in order to have a global distribution of the eigenvectors of the full matrix.

The two first parts in figure 1 show that the spectrum of a matrix of size 17, for example, has a different distribution if we distribute the full matrix or if we distribute separately the even and odd spectra. To obtain a global distribution by concatenating both spectra in each processor, we have to perform a redistribution of the eigenvectors. This is performed minimizing the communications, by sending the last odd eigenvectors from the last processors in order to complete the same number of blocks of the even spectrum in all processors. After performing this redistribution, the eigenvectors location is reported in the lower part of figure 1. We have also to redistribute the eigenvalues to maintain the correspondence during all the iterative process.

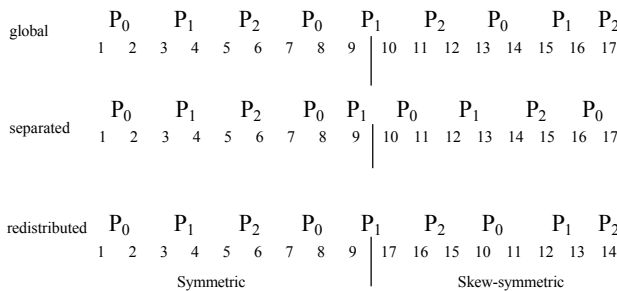


Fig. 1. Global, separated and redistributed odd and even spectra.

The second routine utilized in our parallel algorithm constructs the coefficient matrix of the linear system from the eigenvectors of the TRS matrix. We gather all the eigenvectors in the first row of processors in order to ease the construction of the matrix. Then, we scatter the constructed matrix among all processors in order to solve the system of equations.

Finally, the third and fourth basic routines utilized in our parallel algorithm are the ScaLAPACK routines `PDGETRF` and `PDGETRS`. These routines solve in parallel a general linear system of equations. A complete scheme of computations and communications in the algorithm is reported in figure 2.

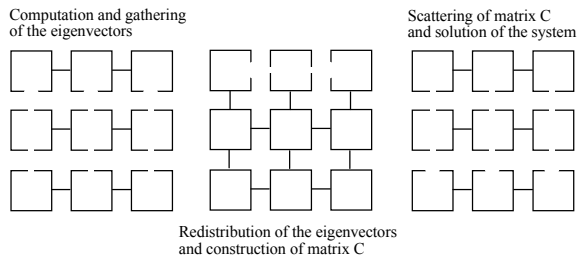


Fig. 2. Computations and communications in the parallel algorithm.

We have performed all communications using the BLACS routines and the auxiliary matrix redistribution routines included in ScaLAPACK, [1]. In both cases all the communications are based on the MPI environment.

4 Experimental Analysis

4.1 Sequential Algorithm

In the case of the sequential version of algorithm 2, we have performed a set of tests similar to the one presented in [6]. In our case the tests have been developed with a Fortran version of the program and with larger matrices ($n=1200$). In all performed tests the algorithm converges, except when the spectrum contains large clusters of eigenvalues. In this case a heuristic convergence strategy is proposed in [6].

We have tested the algorithm with three classes of starting generators t^0 and the best generators are of the class proposed in [6]. Regarding to this, we have observed that using a slight modification of the starting generator used by Trench the convergence always occurs during the quadratic stage of the algorithm or from a value of $\rho=0.1$ obtained during the linear stage.

We have completed the analysis of the sequential case testing the behaviour of the algorithm with 15 classes of spectra with different distributions of the eigenvalues. The algorithm converged in all cases except when large clusters of eigenvalues are present. A more detailed description of the algorithms can be found in [3].

4.2 Parallel Algorithm

First we test the parallel algorithm on a shared memory multiprocessor, the SGI Power Challenge, using 10 MIPS R10000 processors. In this machine the communications are “emulated” on the shared memory, so we cannot fully interpret the results as those of a message-passing model.

Figure 3 shows that the speedups obtained with small matrices are even smaller than 1 for 10 processors. However, executing the algorithm with large matrices we can even achieve superspeedups. This behaviour of the algorithm is due to the large communications overhead involved. With matrices of large dimension, the effect of the communications is reduced with respect to the computational cost, thus allowing better speedups.

We have also studied the enormous influence in the results of the configuration of the mesh. The best results are always obtained with meshes of type $1 \times P$, while poorer results are obtained with meshes of type $P \times 1$. This behaviour of the algorithm is due to its communications scheme. As shown in figure 2, the larger is the number of column processors in the mesh, the smaller is the cost of gathering and scattering the matrices in each iteration of the algorithm.

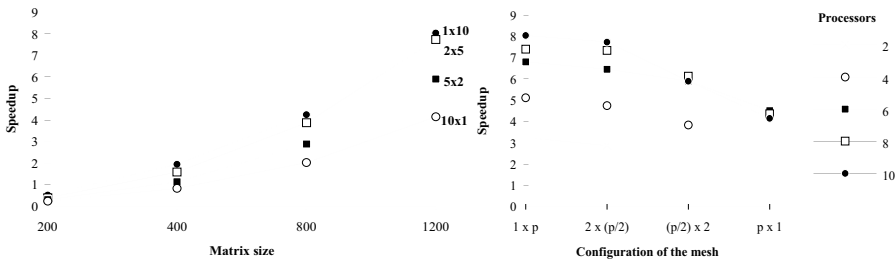


Fig. 3. Speedup vs. matrix size with $p=10$ and Speedup vs. configuration of the mesh with $n=1200$. Results obtained in the Power Challenge.

An experimental analysis has also been performed on a cluster of PCs connected by a Myrinet network and using two different versions of MPI environment. The speedups obtained are smaller than those in the Power Challenge. Notice that in this case we are using an architecture that has a distributed memory and a smaller ratio between computational cost and communications cost. However, in the case of large matrices we also obtain large speedups in this class of architecture.

Finally, in figure 4 we compare the performance obtained using two versions of the MPI environment implemented on the top of a Myrinet network. The GM version has been implemented by the vendor of the network and offers better performances for our problem than the BIP version, implemented in the Laboratory for High Performance Computing in Lyon.

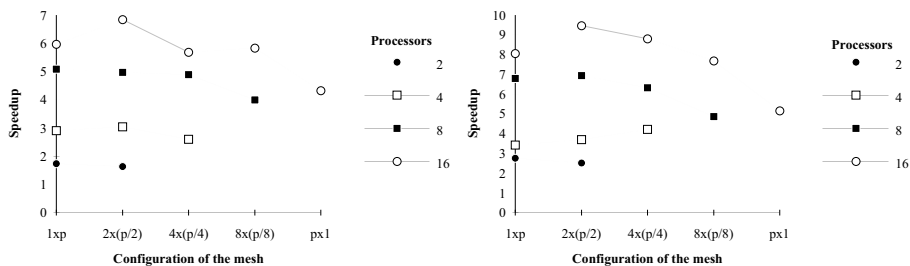


Fig. 4. Evolution of the speedups using different configurations of the mesh ($n=1200$). Results in a cluster of PCs and using two versions of the MPI environment (BIP and GM).

5 Conclusions

In this paper we show a scalable and portable parallel algorithm to solve the inverse eigenproblem of real symmetric Toeplitz matrices. We have obtained an efficient implementation of the algorithm that allows us to solve large problems with a enormous sequential cost. This parallel algorithm can be used to test the convergence of the method using different classes of starting generators and a larger test matrix set.

The results show the convergence of the algorithm in all cases except when the spectrum contains large clusters of eigenvalues. The experimental analysis performed shows scalable speedups, both in a shared memory multicomputer, a SGI Power Challenge, and in a cluster of Pentium II processors connected by a Myrinet network.

The parallelization of the method implies the redistribution of a lot of data in each iteration. The communications cost limits the performance in the case of small matrices. Moreover, we have seen that, given the communications scheme of the algorithm, the best results are obtained with horizontal meshes (1xP), while poorer results are obtained when we increment the number of rows of processors.

References

1. Blackford, L.S., et al.: ScaLAPACK Users' Guide., (1997), Philadelphia: SIAM Press.
2. Snir, M., et al.: MPI. The complete reference. Scientific and Engineering Computation, ed. J. Kowalik. (1996), Cambridge: The MIT Press.
3. Badia, J.M. and A.M. Vidal: Parallel Solution of the Inverse Eigenproblem for Real Symmetric Toeplitz Matrices. (1999). Technical Report DI01-04/99. Dpt. Informàtica, Univ. Jaume I: Castellon.
4. Cantoni, A. and F. Butler: Eigenvalues and eigenvectors of symmetric centrosymmetric matrices. *Lin. Alg. Appl.*, (1976) (13): p. 275--288.
5. Laurie, D.P.: A Numerical Approach to the Inverse Toeplitz Eigenproblem. *SIAM J. Sci. Sta. Comput.*, (1988). 9(2): p. 401--405.
6. Trench, W.F.: Numerical Solution of the Inverse Eigenvalue Problem for Real Symmetric Toeplitz Matrices. *SIAM J. Sci. Comput.*, (1997). 18(6): p. 1722--1736.

A Parallel Implementation of the Eigenproblem for Large, Symmetric and Sparse Matrices.*

E.M. Garzón, I. García

Dpto Arquitectura de Computadores y Electrónica.

Universidad de Almería. 04120-Almería. Spain.

Phone: 34 950215689, 34 950215427. e-mail: ester@iron.ualm.es, inma@iron.ualm.es

Abstract This work studies the eigenproblem of large, sparse and symmetric matrices through algorithms implemented in distributed memory multiprocessor architectures. The implemented parallel algorithm operates in three stages: structuring input matrix (Lanczos Method), computing eigenvalues (Sturm Sequence) and computing eigenvectors (Inverse Iteration). Parallel implementation has been carried out using a SPMD programming model and the PVM standard library. Algorithms have been tested in a multiprocessor system Cray T3E. Speed-up, load balance, cache faults and profile are discussed. From this study, it follows that for large input matrices our parallel implementations perceptibly improve the management of the memory hierarchy.

1 Introduction

This work deals with the eigenproblem of a sparse and symmetric matrix of large dimensions, $A \in \mathbb{R}^{n \times n}$, by means of a sequence of orthogonal transformations. The full process can be summed up in the following mathematical expression:

$$A = Q T Q^T = Q M D M^T Q^T = G D G^T; \quad G = Q M \quad (1)$$

where columns of G are eigenvectors of A , and non zero elements of the diagonal matrix D are eigenvalues of A . This sequence of transformations consists of the following stages: (a) Structuring input matrix, (b) computing eigenvalues, (c) computing eigenvectors of structured matrix T and (d) computing eigenvectors of input matrix A . At stage (a) matrices T and Q are the result of applying the well known Lanczos Method with complete reorthogonalization. At stage (b) a diagonal matrix, D , is the result of applying the so called Sturm Sequence method to the tridiagonal matrix T . At stage (c) Inverse Iteration method has been implemented to compute eigenvectors of T . Eigenvectors of T are the columns of a matrix M , which is computed from matrices T and D (see equation (1)). At stage (d) a matrix product, $G = Q M$, must be computed to get the eigenvectors of A . The columns of the orthogonal matrix G are the eigenvectors of A .

* This work was supported by the Ministry of Education and Culture of Spain (CICYT TIC96-1125-C03-03)

This paper is organized as follows. In Section 2, mathematical foundations of the method applied in this paper are briefly introduced. In Section 3, parallel implementations of all stages are described. Then, in Section 4, results of the performance evaluation of our parallel implementation are shown and discussed. Super-speed-up values are obtained for several large test input matrices, the parallel algorithm is evaluated and the results are justified. Finally, it can be stated that as the number of elemental processors increases the management of memory improves and consequently a good efficiency is achieved.

2 Theoretical Background

The method used to solve the eigenproblem of a large, symmetric, sparse matrix $A \in \mathbb{R}^{n \times n}$, can be described by the following algorithm called EigValVect():

EigValVect($A, \mathbf{D}, \mathbf{G}$)

```

1  SLCR( $A, \mathbf{T}, \mathbf{Q}$ )      # Lanczos with Complete Reorthogonalization #
2  Sturm( $T, \mathbf{D}$ )        # Sturm Sequence#
3  InvIter( $T, D, \mathbf{M}$ )    # Inverse Iteration #
4   $G = Q \ M$            # Matrix  $\times$  Matrix #
```

Input parameters for procedures are denoted by standard signs and output parameters are written in bold. These procedures are discussed in this section.

In order to find a structured matrix similar to input sparse matrix, the Lanczos Method with Complete Reorthogonalization has been used. Given a symmetric matrix $A \in \mathbb{R}^{n \times n}$ and a vector q_0 with unit norm, the Lanczos method generates in n steps an orthonormal matrix Q and a tridiagonal matrix T . This method is discussed in [3, 4, 9, 10]. The outer loop of the Lanczos algorithm is an iterative procedure (index j). At the j -th iterative step α_j and β_j coefficients of the tridiagonal matrix T and vector q_{j+1} are obtained, where α_j and β_j denote elements of the main and secondary diagonals of T , respectively. There is a data dependence which prevents this loop from parallel execution.

The Sturm Sequence is a standard method for determining the eigenvalues of a tridiagonal matrix. This method is discussed in detail in [2, 4, 9, 12]. Let (x, y) be the lower and upper bounds of the starting interval where the k -th eigenvalue of T ($\lambda_k(T)$) is located ($\lambda_k(T) \in (x, y)$). This interval can be determined by the Gershgorin's circle theorem [9].

IterSturm(T, k, x, y)

```

1   $z = x$ 
2  while  $|y - x| > u(|y| + |z|)$ 
3       $x = \frac{(y+z)}{2}$  if  $a(x) \geq k$     $z = x$    else    $y = x$ 
4  return( $\lambda_k(T) = x$ )
```

u is a small positive real number near the rounding error of the machine, and $a(x)$ is the number of sign changes in the sequence $\{P_0(\lambda), P_1(\lambda), \dots, P_n(\lambda)\}$

with $P_0(x) = 1$ and $P_r(x) = \det(T_r - xI)$, $r = 1 \dots n$ and T_r is the principal submatrix of dimension r .

On the other hand, during the execution of $\text{IterSturm}(T, k, x, y)$, additional information about the location of the rest of eigenvalues $(\lambda_l(T), l = k+1, \dots, n-1)$ is obtained. Thus, as the integer l increases, the starting interval for $\lambda_l(T)$ decreases. The $\text{Sturm}(T, \mathbf{D})$ algorithm computes every element of the diagonal matrix D based on the previous considerations.

$\text{Sturm}(T, \mathbf{D})$

```

1  (x, y) = Starting Interval for k = 0 # Gershgorin's circle theorem #
2  do k = 0 ... n - 1
3       $\lambda_k(T) = \text{IterSturm}(T, k)$ 
4      (x, y) = Starting Interval for k = k + 1 # additional information #
```

Given any scalar λ_k which is a good approximation to a simple eigenvalue of T , the Inverse Iteration method provides a mechanism for computing an approximation to the corresponding eigenvector of T . The algorithm $\text{InvIter}(T, D, \mathbf{M})$ is based on this method.

$\text{InvIter}(T, D, \mathbf{M})$

```

1  do k = 0 ... n - 1
2       $m_k = (1, \dots, 1)$ ; threshold =  $u \| T \|_\infty$ 
3      while (error  $\leq$  threshold)
4           $m_k = m_k u \| T \|_\infty$ 
5          Solver  $(T - \lambda_k I)z_k = m_k$ 
6           $m_k = \frac{z_k}{\|z_k\|}$ 
7          error =  $\| (T - \lambda_k I)m_k \|_\infty$ 
```

m_k denotes the column k of matrix M . Details about convergence criteria and scaling of starting vector for right term in linear system can be found in [4, 9, 12].

3 Parallel Algorithms

The above discussed method to solve eigenproblem of a symmetric matrix has a extremely high computational cost. Furthermore, this method demands large memory requirements. Consequently, its implementation on a distributed memory multiprocessor is specially appropriate.

Parallel implementation of $\text{EigValVect}()$ has been carried out using a SPMD programming model and the PVM standard library. The whole solution of the eigenproblem ($\text{EigValVect}()$) has been divided into a set of procedures: $\text{SLCR}()$, $\text{Sturm}()$, $\text{InvIter}()$, Matrix-Matrix Product. These link their executions in a sequential way, because data dependences prevent several procedures from the simultaneous execution. Thus, every procedure must be independently parallelized.

In the whole eigenproblem, only $\text{SLCR}()$ processes irregular data structures. i.e., sparse input matrix A . Since the Lanczos algorithm works on mixed computations (dense-sparse) a special care must be taken in the data distribution among processors in order to optimize the work load balance for computations on both dense and sparse data structures. A data distribution called *Pivoting Block* [8] has been designed to balance the computational load of this procedure. Details about parallel implementation of $\text{SLCR}()$ can be found in [6, 7]. The output data of this parallel algorithm are: the tridiagonal matrix T and the orthonormal matrix Q . Whole T is stored at local memory of every Processing Element (PE). When the parallel procedure $\text{SLCR}()$ finishes, every PE stores at its local memory $\lceil \frac{n}{P} \rceil$ rows of Q , where P is the number of PEs in the multiprocessor system.

Several papers have already dealt with the parallelization of bisection and multisection methods using the Sturm Sequence [1, 11]. In our parallel implementation, iterations of the outer loop of $\text{Sturm}()$ (index k) are distributed among PEs. Thus, every PE labeled by integer I compute $\lceil \frac{n}{P} \rceil$ consecutive eigenvalues by the following algorithm:

$\text{ParSturm}(T, \text{Dloc})$

```

1   Starting Interval for  $k = \lceil \frac{n}{P} \rceil$ ,  $(x, y)$  # Gershgorin's circle theorem #
2       do  $k = \lceil \frac{n}{P} \rceil \times I, \dots, \lceil \frac{n}{P} \rceil \times (I + 1) - 1$ 
3            $\lambda_k(T) = \text{IterSturm}(T, k, x, y)$ 
4       Starting Interval for  $k = k + 1$ ,  $(x, y)$  # additional information #
```

The light data dependence in the $\text{Sturm}()$ procedure allows to compute simultaneously several iterations of the outer loop. Only a light data dependence exists because the starting interval to seek the eigenvalue $\lambda_k(T)$ is reduced by the previous computation of $\lambda_l(T)$ with $l = 0, \dots, k$. Nevertheless, in our parallel implementation of the Sturm Sequence Method every PE computes a set of eigenvalues without interprocessor communications. The starting interval to seek the eigenvalue $\lambda_k(T)$ is reduced only by the inner computation of $\lambda_l(T)$ ($l = \lceil \frac{n}{P} \rceil \times I \dots k$) on every PE. Thereby, this dependence is kept up to processed data by every PE and the number of iterations to compute the first eigenvalues at every PE is slightly increased. It has been checked that this penalty is negligible, due to the good convergence of this method.

$\text{ParSturm}()$ can produce a light work load unbalance because the method used to compute every eigenvalue (IterSturm) is iterative and the number of iterations needed to compute every eigenvalue may change. However, when the dimension of T is large, the total number of iterations executed by every PE is similar for all the PEs, so the global unbalance is negligible.

Parallel implementation of the Inverse Iteration method is similar to the Sturm Sequences Method. When $\text{ParSturm}()$ finishes a set of $\lceil \frac{n}{P} \rceil$ good approximations of consecutive eigenvalues (Dloc) and the full matrix T are stored at local memories. Thus, every PE has the input data to compute $\lceil \frac{n}{P} \rceil$ eigenvectors by Inverse Iteration method. The computation of every eigenvector is independent.

Thereby the parallel implementation of this method doesn't include interprocessor communications.

Moreover, it is necessary to compute the matrix G by the matrix-matrix product, $Q \times M$, and it is necessary a very large memory to store these matrices. To reduce memory requirements, when m_k is computed by Inverse Iteration method, the column k of G (g_k) is computed by the matrix-vector product $Q \times m_k$. Thus, only one column of the matrix M has to be stored at the local memory of every PE, but in this case the penalty due to interprocessor communications increases. To clearly explain parallelization of $G = Q M$, Figure 1 has been drawn. Thus,

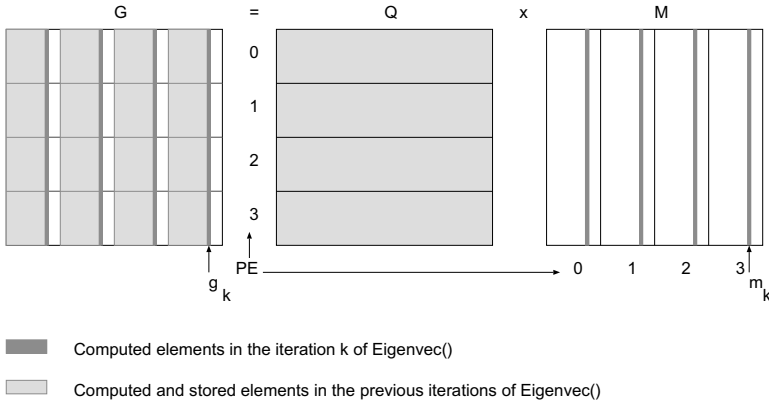


Figure1. Parallel Implementation of matrix-matrix product $G = Q \times M$ with $P = 4$.

the parallel algorithm to compute the columns of G can be explained as follows:

```

Eigenvec( $T, Dloc, Qloc, Gloc$ )
1   do  $k = 0, \dots, \lceil \frac{n}{P} \rceil$ 
2        $x = y_k = InvIter(T, \lambda_k)$ 
3        $G_l[k + I \times \lceil \frac{n}{P} \rceil] = Qloc \times x$ 
4       do  $j = 1, \dots, P$ 
5           Send  $x$  to  $(I + 1) \bmod P$ 
6           Recv  $x$  from  $(I - 1) \bmod P$ 
7        $Gloc[(k + (I - j) \times \lceil \frac{n}{P} \rceil) \bmod n] = Qloc \times x$ 

```

4 Evaluation of Parallel Algorithms

The evaluation of parallel implementation of the eigenproblem has been carried out on a multiprocessor system Cray T3E using a set of three input matrices with dimension n and percentage of non-zero elements s . These matrices belong to the Harwell-Boeing collection and NEP collection of test matrices [5]. These

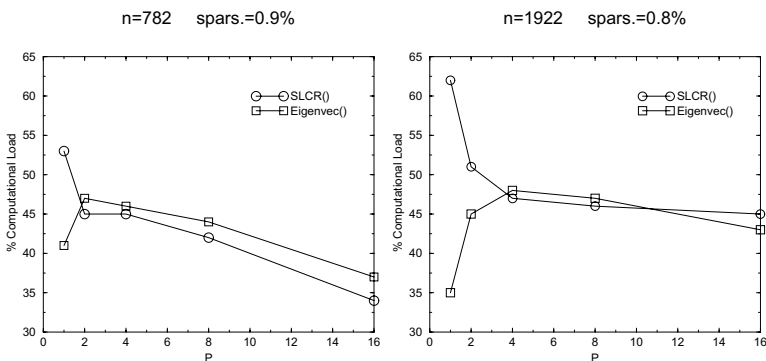
Table1. Test Matrices and accuracy.

Matriz	n	s	$\log_{10} \frac{R}{\ A\ _F}$
BFW782B	782	0.9 %	-15
BCSSTK27	1224	3.0 %	-14
BCSSTK26	1922	0.8 %	-14

parameters for several test matrices are shown in Table 1. The third column of Table 1 shows the accuracy of solutions for several test matrices by the magnitude order of values for parameter $\log_{10} \frac{R}{\|A\|_F}$, where R is the norm of the residual related to the computed solution and it is defined as $R = \|AG - GD\|_F$ and $\|\cdot\|_F$ denotes the Frobenius matrix norm.

Values up to linear speed-up have been obtained in most of the analyzed cases. This fact is more evident for large input matrices. Our aim has been to justify these results. Thus, the parallel implementation has been analyzed through the following set of additional parameters: number of executed operations by every PE; number of cache faults and distribution of the computational work among the set of procedures involved in the parallel algorithm (algorithm profile). Experimental results will show that the management of the memory hierarchy plays an important role in algorithms with large memory requirements.

For all the studied cases, a good work load balance has been obtained. On the other hand, the experimental profile data for this algorithm indicate that the SLCR() and Eigenvec() procedures consume between 71% and 97% out of the total computational burden. The greatest percentage (97%) were obtained for smaller values of P , since in this case the penalty of communications is less considerable. In Figure 2 percentages of the total computational work associated with SLCR() and Eigenvec() procedures have been plotted for two test matrices. As can be observed in Figure 2, the percentage of the computational work asso-


Figure2. Percentages of the total computational burden associated with SLCR() and Eigenvec() procedures.

ciated with `SLCR()` is ostensibly greater than the one associated with `Eigenvec()` when it is executed with very few PEs. But as the number of PEs increases, the cost associated with both procedures is balanced. Thus, the balance for both procedures is achieved by $n = 782$ when $P = 2$, and by $n = 1922$ when $P = 4$. The global cost of both procedures is reduced as P increases, due to the increase in communications time, especially for small input matrices.

Experimentally we have obtained the number of cache faults against P and it can be seen that the number of cache faults reduces considerably as P increases. When the data access is irregular, the probability of cache faults is greater than when this is regular. The irregularity of this problem is connected to `SLCR()`. This fact justifies the behaviour of algorithm profile (see Figure 2).

Speed-up values against the number of processors, P , have been plotted on the left in Figure 3. Speed-up value is higher as the input matrix dimension increases. The sequential executions are slower, not only because there is just one PE, but also, because the data access is slower. The values of the speed-up are higher than the linear values for every discussed case, because this parameter is obtained as the quotient from the run time in the sequential execution and the run time in a execution with P PEs. Thus, the penalty of the data access time from the sequential execution remains in every value of speed-up. The parameter defined by us and called Incremental Speed-up (**SpUpInc**) does not reflect this penalty for all the values of P . **SpUpInc** is defined as follows:

$$SpUpInc(2^i) = \frac{T(P = 2^{i-1})}{T(P = 2^i)} \quad (2)$$

where $T(P)$ is the run time of a execution with P PEs. For ideal parallel implementations, the value of this parameter should be equal to 2. On the left of

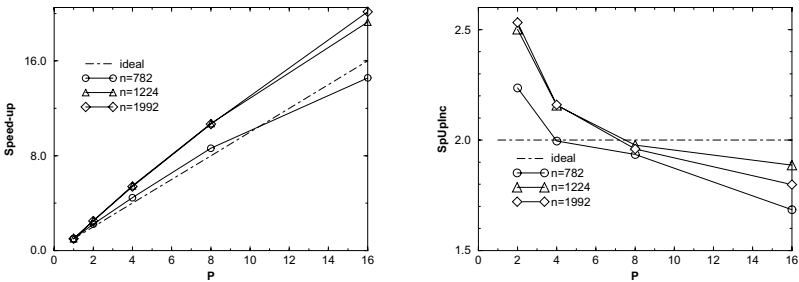


Figure3. On the left, Speed-up for several dimensions of input matrix. On the right, Incremental Speed-up for several dimensions of input matrix.

Figure 3, Incremental Speed-up has been plotted, for three test matrices. The behavior of the parallel implementation can be analyzed by this plot, for all the

values of P . The decrease of the cache faults and the penalty of communications are clearly shown by the parameter $SpUpInc$. For the set of test matrices, the value of $SpUpInc$ is greater than 2 for the first values of $P = 2^i$. This fact is due to a decreasing of the cache faults. As P increases this improvement vanishes and the penalty of interprocessor communications increases obtaining values of $SpUpInc$ less than the ideal value.

5 Conclusions

For large matrices, large memory requirements are needed by the implemented algorithm, and the data access is irregular for a data subset. Consequently, the management of memory hierarchy is very relevant for this kind of algorithms. The designed parallel implementation is able to distribute in a balanced way the computational work load associated with all the procedures; to establish interprocessor communications that do not increase considerably the run time, and what is more, to improve the data access time, especially for irregular data.

References

1. Basermann, A.; Weidner, P. *A parallel algorithm for determining all eigenvalues of large real symmetric tridiagonal matrices*. Parallel Computing 18, pag 1129-1141. 1992.
2. Bernstein, H.J. *An Accelerated Bisection Method for the Calculation of Eigenvalues of a Symmetric Tridiagonal Matrix*. Numer. Math. 43, pag. 153-160. 1984.
3. Berry, M.W. *Large-Scale Sparse Singular Value Computations*. The International Journal of Supercomputer. Vol 6, pg. 171-186. 1992.
4. Cullum, J.K. and Willoughby, R.A. *Lanczos Algorithms for Large Symmetric Eigenvalue Computations. Vol. I Theory*. Birkhauser Boston, Inc., 1985.
5. Duff, I.S.; Grimes, R.G. and Lewis, J.G. *User's Guide for the Harwell-Boeing Sparse Matrix Collection*. Research and Technology Division, Boeing Computer Services. 1992.
6. García, I; Garzón, E.M.; Cabaleiro, J.C.; Carazo, J.M. and Zapata, E.L. *Parallel Tridiagonalization of Symmetric Matrices Based on Lanczos Method*. Parallel Computing and Transputer Applications. Vol I, 236-245, 1992.
7. Garzón, E.M. and García, I. *Parallel Implementation of the Lanczos Method for Sparse Matrices: Analysis of the Data Distributions*. Proceedings ICS'96. p. 294-300. Philadelphia, 1996.
8. Garzón, E.M. and García, I. *Evaluation of the Work Load Balance in Irregular Problems Using Value Based Data Distributions*. Proceedings of Euro-PDS'97. p. 137-143. 1997.
9. Golub, G.H. and Van Loan C. F. *Matrix Computation*. The Johns Hopkins University Press., Baltimore and London., 1993.
10. Simon, H.D. *Analysis of the symmetric Lanczos Algorithm with Reorthogonalization Methods*. Linear Algebra and its Applications. Vol 61, pg. 101-131. 1984.
11. Sy-Shin Lo, Bernard Philippe and Ahmed Samed. *A multiprocessor Algorithm for the Symmetric Tridiagonal Eigenvalue Problem*. SIAM J. Sci. Stat. Comput. vol-8, No. 2, March 1987.
12. Wilkinson, J.H. *The Algebraic Eigenvalue Problem*. Clarendon Press. Oxford, 1965.

Parallel Computation of the SVD of a Matrix Product ^{*}

José M. Claver¹, Manuel Mollar¹, and Vicente Hernández²

¹ Dpto. de Informática, Univ. Jaume I, E-12080 Castellón, Spain

² Dpto. de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia, E-46071 Valencia, Spain

Abstract. In this paper we study a parallel algorithm for computing the singular value decomposition (SVD) of a product of two matrices on message passing multiprocessors. This algorithm is related to the classical Golub-Kahan method for computing the SVD of a single matrix and the recent work carried out by Golub *et al.* for computing the SVD of a general matrix product/quotient. The experimental results of our parallel algorithm, obtained on a network of PCs and a SUN Enterprise 4000, show high performances and scalability for large order matrices.

1 Introduction

The problem of computing the singular value decomposition (SVD) of a product of matrices occurs in a great variety of problems of control theory and signal processing (see [14,15,18]). The product singular value decomposition (PSVD) is defined as follows.

For any given matrices, $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$, there exist two matrices $U \in \mathbb{R}^{m \times n}$ and $V \in \mathbb{R}^{n \times p}$ with orthogonal columns, an orthogonal matrix $Q \in \mathbb{R}^{n \times n}$, and a full rank upper triangular matrix $R \in \mathbb{R}^{n \times n}$, such that

$$A = U \Sigma_A R Q^T, \quad B = Q R^{-1} \Sigma_B V^T \quad (1)$$

with

$$\Sigma_A = \text{diag}(\alpha_1, \dots, \alpha_n), \quad \Sigma_B = \text{diag}(\beta_1, \dots, \beta_n). \quad (2)$$

The n pairs (α_i, β_i) are called the product singular values of (A, B) and the singular values of the product AB are the products $\alpha_i \beta_i$, for $i = 1, \dots, n$.

The PSVD was introduced by Fernando and Hammarling [8] as a new generalization of the SVD, based on the product AB^T . It complements the generalized singular value decomposition (GSVD) introduced by Van Loan [20] and Paige and Sanders [19], now called quotient-SVD (QSVD), as proposed in [5]. The parallel computation of the GSVD has been treated in [2,3]. A complete study of the properties of the PSVD can be found in [6]. The computation of the PSVD is typically carried out with an implicit Kogbetliantz algorithm, as proposed in [8]. Parallel implementations of this algorithm are presented in [16,17].

^{*} This research was partially supported by the Spanish CICYT project under grant TIC96-1062-C03-01-03.

In this work we are interested in the computation of the SVD of the product AB . The method used is based on the algorithm designed by Golub *et al.* in [12]. Their algorithm is related to the Golub-Kahan procedure for computing the singular value decomposition of a single matrix in that a bidiagonal form of the sequence, as an intermediate result, is constructed [10]. The algorithm derived in [12] applies this method to two matrices as an alternative way of computing the product SVD. In this paper we describe a parallel algorithm for computing the singular values of the product of two matrices and study its implementation on two different message passing multiprocessors using ScaLAPACK.

The rest of the paper is organized as follows. Section 2 describes the implicit bidiagonalization as first step of this method. In section 3 we present a sequential version of this algorithm using LAPACK and the characteristics of our ScaLAPACK based parallel algorithm. In section 4 we analyze the performance and scalability of our parallel algorithm on both shared and distributed memory multiprocessors.

2 Implicit Bidiagonalization

Given two matrices A and B , we want to compute the SVD of the product AB without explicitly constructing of their product. This method involves two stages:

1. The implicit bidiagonalization of the product AB .
2. The computation of the SVD of the bidiagonal matrix.

For the first stage we need about $O(n^3)$ flops; for the second stage we need about $O(n^2)$ flops. Well-known Golub-Reinsch [11], Demmel-Kahan [7] or Fernando and Parlett [9] algorithms can be employed to perform the second stage. Thus, we have focused our efforts on the first stage.

For this purpose, we generate a sequence of matrix updates with the application of different Householder transformations [13]. In order to illustrate the procedure we show its evolution operating on a product of two 4x4 matrices A and B . Here, x stands for a nonzero element, and 0 a zero element.

First, we perform a Householder transformation Q_0 on the rows of B and the columns of A , chosen to annihilate all but the first component of the first column of B :

$$\begin{aligned} AB &= AQ_0^T Q_0 B, \\ A &\leftarrow AQ_0^T, \\ B &\leftarrow Q_0 B, \end{aligned} \quad A = \begin{bmatrix} x & x & x & x \\ x & x & x & x \\ x & x & x & x \\ x & x & x & x \end{bmatrix}, \quad B = \begin{bmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \end{bmatrix}.$$

Then, we perform a Householder transformation Q_A on the rows of A , chosen to annihilate all but the first component of the first column of A :

$$Q_A A = \begin{bmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \end{bmatrix}$$

Notice that the resulting matrix product $Q_A AB$ presents the same structure:

$$\begin{bmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \end{bmatrix} \begin{bmatrix} x & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \end{bmatrix} = \begin{bmatrix} \mathbf{x} & \mathbf{x} & \mathbf{x} & \mathbf{x} \\ 0 & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \end{bmatrix}.$$

We are interested in the first row of this product (indicated in boldface above). This row can be constructed as the product of the first row of A and the matrix B . Once it is constructed, we find a Householder transformation operating on the last $(n - 1)$ elements of the row, which annihilates all but the two first components of of this row:

$$A(1,:)BQ_B = [x \ x \ 0 \ 0].$$

Thus, when this transformation is applied to B , the first step of the bidiagonalization is completed as

$$Q_A ABQ_B = \begin{bmatrix} x & x & 0 & 0 \\ 0 & x & x & x \\ 0 & x & x & x \\ 0 & x & x & x \end{bmatrix}.$$

Next, we consider $A(2 : n, 2 : n)$ and $B(2 : n, 2 : n)$, and using the same procedure, we bidiagonalize the second column/row of the matrix product. The procedure is repeated until the full matrix product is bidiagonalized.

3 Parallel Algorithm

We have implemented a BLAS-2 routine (denoted by BD) for computing implicitly the bidiagonal of product of two $n \times n$ matrices using the LAPACK library [1].

Algorithm 1 [BD]

for $i = 1, 2, \dots, n - 1$

Compute the Householder reflector that nullify $B(i+1:n,i)$:	DLARFG
Apply the Householder reflector to B from the left:	DLARF
Apply the reflector to A from the Right:	DLARF
Compute the reflector that nullify $A(i+1:n,i)$:	DLARFG
Apply the reflector to A from the left:	DLARF
Compute the implicit product AB ($A(i,i:n)B(i:n,i:n)$):	DGEMV
Compute the reflector that nullify $AB(i,\min(i+2,n):n)$:	DLARFG
Store diagonal and off-diagonal elements of the matrix product	
Apply the reflector to B from the right:	DLARF

end for

Last diagonal element of the matrix product is $A(n,n)B(n,n)$

Algorithm 1 shows, in a detailed manner, the procedure followed by our *BD* routine, where the LAPACK routines used are described to the right hand of each step. The routine obtains two vectors corresponding to the diagonal and the upperdiagonal of the bidiagonalized matrix product.

In order to obtain the SVD of the resulting bidiagonal we can use either the xLASQ or the xBDSQR LAPACK routines, that implement the Fernando-Parlett[9] and the Demmel-Kahan [7] algorithms, respectively.

Our parallel implementation (denoted by *PBD*), is implemented in ScaLAPACK [4] and is presented in algorithm 2, that includes the corresponding ScaLAPACK routines used. In this parallel library, the matrices are block cyclically distributed among a $P = p \times q$ mesh of processors.

Algorithm 2 [*PBD*]

Input:

$A, B \in \mathbb{R}^{n \times n}$: distributed matrices

Output:

$D, E \in \mathbb{R}^n$: local vectors to store the fragments of the resulting bidiagonal

Create ScaLAPACK descriptors for D and E

for $i = 1, 2, \dots, n - 1$

Compute the Householder reflector that nullify $B(i+1:n,i)$:	PDLARFG
Apply the Householder reflector to B from the left:	PDLARF
Apply the reflector to A from the Right:	PDLARF
Compute the reflector that nullify $A(i+1:n,i)$:	PDLARFG
Apply the reflector to A from the left:	PDLARF
Store diagonal ($D(i) \leftarrow B(i, i)$) element:	PDELSET
Compute the implicit product AB ($A(i,i:n)B(i:n,i:n)$):	PDGEMV
using part of D as a resulting distributed vector	
Store off-diagonal ($E(i)$) element:	PDELSET
Compute the reflector that nullify $AB(i, \min(i+2, n):n)$:	PDLARFG
Apply the reflector to B from the right:	PDLARF

end for

Last diagonal element of the matrix product is $A(n,n)B(n,n)$

4 Experimental Results

In this section we analyze the performance of our parallel algorithm obtained on two different architectures, a SUN Enterprise 4000 and a network of PCs(Personal Computers). We compare the performance of the serial and the parallel routines on the SUN and the PC cluster. The test matrices are full ones randomly generated.

The SUN 4000 is a shared memory non bus-based multiprocessor (the main memory has 512 MBytes) with 8 UltraSparc processors at 167 MHz and a second

level cache of 512 KBytes. It has a 4×4 crossbar interconnection network between pairs of processors and main memory.

The network of PCs consists of 16 PCs interconnected by a Myricom Myrinet network (MMn). Each PC is a Pentium II at 300 MHz, with 512 KBytes of second level cache memory and 128 MBytes SDRAM of local memory per processor, under Linux operating system. The Miricom Myrinet is an 8×8 bidirectional crossbar network with a bandwidth of 1.28 Gbits/s per link.

All experiments were performed using Fortran 77 and IEEE double-precision arithmetic. We made use of the LAPACK library and the ScaLAPACK parallel linear algebra library [4]. The use of these libraries ensures the portability of the algorithms to other parallel architectures.

The communication in the ScaLAPACK library is carried out using the MPI communications library. We have used optimized MPI libraries on both the SUN (vendor supplied) and the MMn (GM version) machines.

Our first experiment is designed to evaluate the efficiency of our parallel algorithm *PDB*. In Figures 1 and 2 we report the efficiencies obtained for our algorithm, using $P = 2, 4, 6$ and 8 processors, on the MMn and the SUN platforms, respectively. Notice that high performances are obtained for medium-scale and large-scale problems on both machines. On the SUN, the efficiency of our parallel algorithm grows more quickly than in the MMn machine when the order of the matrices is increased. The reason is the better relative speed of communication on the SUN.

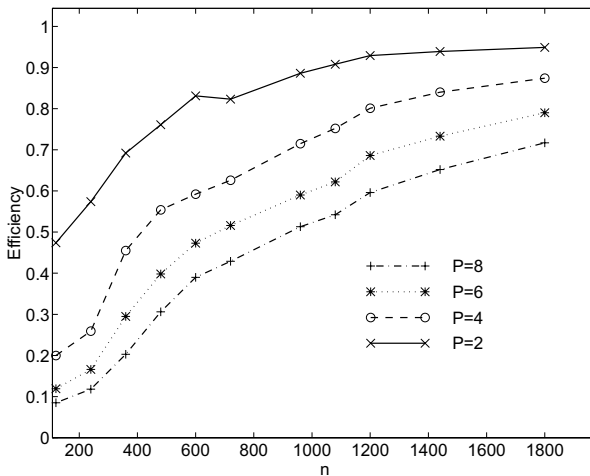


Fig. 1. Efficiencies obtained on the MMn using $P = 2, 4, 6$ and 8 processors for different problem orders.

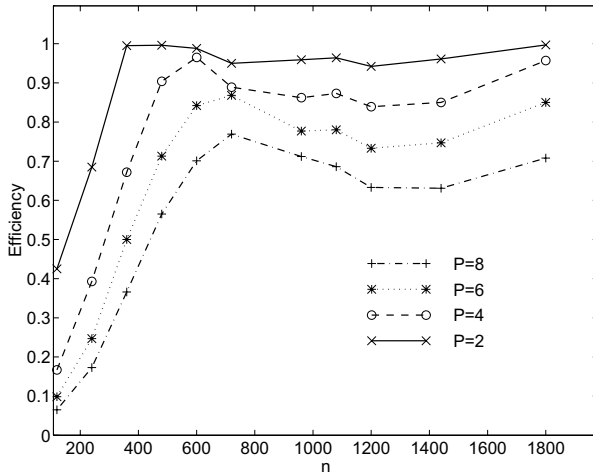


Fig. 2. Efficiencies obtained on the SUN using $P = 2, 4, 6$ and 8 processors for different problem orders.

In our following experiment we analyze the scalability of the parallel algorithm on the MMn platform. In Figure 3 we fix the memory requirements per node of the algorithm to $n/p = 500, 1000, 1500$ and 2000 , and report the megaflops per node for $P = 1 \times p^2 = 1, 4, 9$, and 16 processors. Notice that the performance is slightly degraded as the number of processors gets larger, and this performance degradation is minimum as the memory requirements per node are increased.

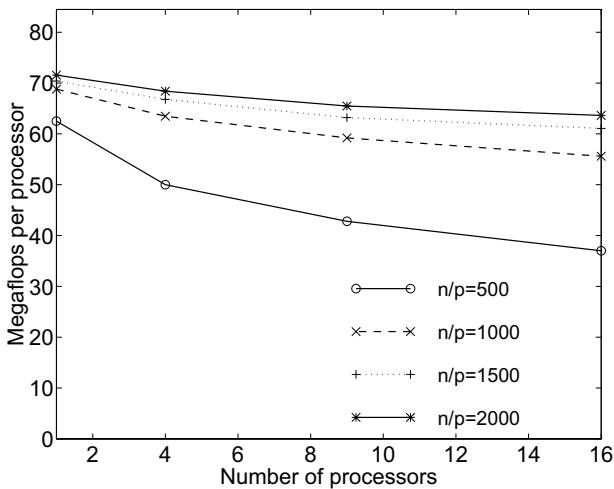


Fig. 3. Mflops per processor obtained on the MMn for constant data load n/p .

5 Concluding Remarks

We have studied the parallelism of a new method for computing the SVD of a matrix product via the implicit bidiagonalization of this product as intermediate step.

The experimental results show the high efficiency of our algorithm for a wide range of scale problems on shared and distributed memory architectures. Moreover, this algorithm shows an excellent scalability on a PC cluster.

Our algorithm can be easily extended for computing the SVD of a general matrix product. Currently, we are working on a BLAS-3 routine in order to increase the performance of both serial and parallel algorithms.

References

1. Anderson, E., Bai, Z., Bischof, C., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., Mckenney, A., Ostrouchov, S., Sorensen, D.: LAPACK User's Guide, Release 1.0., SIAM, Philadelphia (1992).
2. Bai, Z: A parallel algorithm for computing the generalized singular value decomposition, *Journal of Parallel and Distributed Computing* **20** (1994) 280-288.
3. Brent, R., Luk, F. and van Loan, C.: Computation of the generalized singular value decomposition using mesh connected processors, *Proc. SPIE Vol. 431, Real time signal processing VI* (1983) 66-71.
4. Blackford, L., Choi, J., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, L., Hammarling, S., Henry, G., Petitet, A., Stanley, K., Walker, D., Whaley, R.: SCALAPACK User's Guide, SIAM (1997).
5. De Moor, B. and Golub, G. H.: Generalized singular value decompositions: A proposal for a standardized nomenclature, *Num. Anal. Proj. Report 89-04*, Comput. Sci. Dept., Stanford University (1989).
6. De Moor, B.: On the structure and geometry of the PSVD, *Num. Anal. Project, NA-89-05*, Comput. Sci. Dept., Stanford University (1989).
7. Demmel, J., Kahan, W.: Accurate singular values of bidiagonal matrices, *SIAM J. Sci. Stat. Comput.* **11** (1990) 873-912.
8. Fernando, K. and Hammarling, S.: A generalized singular value decomposition for a product of two matrices and balanced realization, *NAG Technical Report TR1/87*, Oxford (1987).
9. Fernando, K. and Parlett, B.: Accurate singular values and differential qd algorithms. *Numerische Mathematik* **67** (1994) 191-229.
10. Golub, G., W. Kahan, Calculation of the singular values and the pseudoinverse of a matrix, *SIAM J. Numer. Anal.* **2** (1965) 205-224.
11. Golub, G., Reinsch, W.: Singular value decomposition and the least square solution, *Numer. Mathematik* **14**, (1970) 403-420.
12. Golub, G., Sølna, K. and van Dooren, P.: Computing the SVD of a General Matrix Product/Quotient, submitted to *SIAM J. on Matrix Anal. & Appl.*, (1997).
13. Golub, G., Van Loan, C.: *Matrix Computations*, North Oxford Academic, Oxford (1983).
14. Heat, M., Laub, A., Paige, C., Ward, R.: Computing the singular value decomposition of a product of two matrices, *SIAM J. Sci. Stat. Comput.* **7** (1986) 1147-1159.
15. Laub, A., Heat, M., Paige, G., Ward, R.: Computation of system balancing transformations and other applications of simultaneous diagonalization algorithms, *IEEE Trans. AC* **32** (1987) 115-122.

16. Mollar, M., Hernández, V.: Computing the singular values of the product of two matrices in distributed memory multiprocessors, Proc. 4th Euromicro Workshop on Parallel and Distributed Computation, Braga (1996) 15-21.
17. Mollar, M., Hernández, V.: A parallel implementation of the singular value decomposition of the product of triangular matrices, 1st NICONET Workshop, Valencia (1998)
18. Moore, B.: Principal component analysis in linear systems: Controlability, observability, and model reduction, IEEE Trans. AC **26** (1981) 100-105.
19. Paige, C., Sanders, M.: Towards a generalized singular value decomposition, SIAM J. Numer. Anal. **18** (1981) 398-405.
20. Van Loan, C.: Generalizing the singular value decomposition, SIAM J. Numer. Anal. **13** (1976) 76-83.

Porting generalized eigenvalue software on distributed memory machines using systolic model principles

Pierre Bassomo¹ and Ibrahima Sakho² and Annie Corbel¹

¹ Equipe RIM (SIMADE) Ecole des mines, Saint-Etienne, France

² LRIM, Université de Metz, Metz, France

Abstract. This paper deals with the issue of porting sequential application on distributed memory machines. Three constraints direct the selection of the parallelization paradigm: little rewriting of the non-parallel source code, portability on several distributed memory machines and the overheads due to data and computation partitioning. The parallelization paradigm found to be most effective for our software on distributed memory machine was to provide the user with a *client/server* architecture.

1 Introduction

The parallelization of existing sequential software for distributed memory machines is a difficult task. Nowadays, distributed memory machines are implemented upon *MIMD* parallel computers or networks of workstations. Such machines are easily scalable, in the means of the number of processor, the amount of memory and the bandwidth of the underlying communication network.

In the area of parallelizing existing sequential application, two main approaches have been proposed: *automatic parallelization techniques* and *explicit parallelization*. Since many scientific applications are written in *Fortran*, automatic parallelization strategies focuses on *HPF* style programming. *HPF* is a language definition that allows to use statements directives and constructs so that the compiler can generate efficient parallel code. Unfortunately, as complex and irregular data structures are to be handled in many applications, automatic parallelization techniques do not always yield efficient parallel codes.

In the present paper, we focus our studies on the second approach. We show how using *systolic algorithm* principles help us in the design of parallel control structures needed in a task-parallel programming model. Such a programming model looks like a “*client/server*” architecture in which the “*client*” executes the non-parallel source code (with some few rewriting) and the *server* is an “*SPMD*” code which executes the parallel the compute-intensive kernels of an application.

The plan of the paper is as follows. In section 2, we present the scientific application that we propose to parallelize. In section 3, we describe the distributed memory software. In section 4, we report some experimental results.

2 The serial application

The serial application to parallelize is a software dealing with computing the solution of the symmetric generalized eigenvalue problem:

$$\begin{cases} \text{find } (x, \lambda) \text{ verifying } x \neq 0 \text{ and} \\ Kx - \lambda Mx = 0 \end{cases} \quad (1)$$

where K and M are real, symmetric and either K or M is positive semi-definite. For the needs of the implementation, K and M are n -by- n sparse matrices. In order to take advantage of the symmetry and to allow some specific operations concerning sparse matrices like the sum and the matrix factorization, K and M are stored in a skyline format and their storage schemes are the same.

In the section 4, we have illustrated some test problems. One needs to compute a few number of eigenvalues of equation 1. Since K and M are symmetric, the *block Lanczos* for generalized eigenvalue problem [Cha88] has shown to allow easy computation of eigenvalue of multiplicity up to r , where r is the number of vectors per *Lanczos block*. The serial *block Lanczos* scheme implemented in the software uses a spectral transformation (see [Cha88]). In order to describe this *Lanczos* process to solve 1, we assume the following transformation: first, let $K_\sigma = K - \sigma M$ and

$$K - \lambda M = K_\sigma(I - (\lambda - \sigma)K_\sigma^{-1}M) \quad (2)$$

and second, let $\mu = 1/(\lambda - \sigma)$ and $S = K_\sigma^{-1}M$. Finally, one can show that solving 1 is equivalent to solve:

$$\begin{cases} \text{find } (y, \mu) \text{ verifying } y \neq 0 \text{ and} \\ Sy - \mu y = 0 \end{cases} \quad (3)$$

The *block Lanczos* procedure is an algorithm for building an orthonormal basis of the *Krylov* subspace:

$$\{U^{(0)}, SU^{(0)}, \dots, S^{l-1}U^{(0)}\} \quad (4)$$

where $U^{(0)}$ is a n -by- r matrix defined by $U^{(0)} = (u_1^{(0)}, \dots, u_r^{(0)})$. The major steps implemented in the present software are the following:

- step 1 : Compute the factorization $K_\sigma = LDL^T$;
- step 2 : Compute the residual $R^{(0)} = (r_1^{(0)}, \dots, r_r^{(0)})$;
- step 3 : For $i = 1, \dots, l$ do:
- step 3.1 : Compute the matrices G_{i-1} , r -by- r and $Q^{(i)}$, n -by- r verifying:

$$\begin{cases} R^{(i-1)} = Q^{(i)}G_{i-1} \\ (MQ^{(i)})^T Q^{(i)} = I_r \end{cases} \quad (5)$$

(The *Gramm-Schmidt* factorization)

- step 3.2 : Compute the full orthogonalization of $Q^{(i)}$ with respect to $Q^{(j)}$, for $j < i$ and the scalar scalar product $(\cdot|\cdot)_M$;

- step 3.3 : Compute $R^{(i)} = (r_1^{(i)}, \dots, r_r^{(i)})$ such that $R^{(i)} = K_\sigma^{-1} M Q^{(i)}$;
 step 3.4 : Compute the norm $\|r_j^{(i)}\|_M$, for $j = 1, \dots, r$ where $\|r_j^{(i)}\|_M^2 = (M r_j^{(i)})^T r_j^{(i)}$;
 step 3.5 : Compute the full orthogonalization of $R^{(i)}$ with respect to $Q^{(j)}$, for $j < i$
 and the scalar scalar product $(\cdot|\cdot)_M$;
 step 3.6 : Compute the $r - by - r$ matrix $A_i = (M Q^{(i)})^T R^{(i)}$.

The steps 3.1 – 3.6 allow a block tridiagonal matrix T_i , describe in figure 1.

$$T_i = \begin{pmatrix} A_1 & G_1^T & 0 & . & . & 0 \\ G_2 & A_2 & G_2^T & 0 & . & 0 \\ 0 & G_3 & A_3 & G_3^T & 0 & 0 \\ . & . & . & . & . & . \\ . & . & . & . & . & G_{i-1}^T \\ . & . & . & . & G_i & A_i \end{pmatrix}$$

Fig. 1. The block tridiagonal matrix T_i after i iteration of the block Lanczos process.

3 The distributed memory software

The parallelization of the software focuses on the following three computations: $K_\sigma = LDL^T$, $y = Mx$ and $K_\sigma^{-1}x = b$. Observation of the runtime profile are reported in section 4. Three constraints directed the selection of the approach to be used in carrying on the parallelization: few rewriting of the non-parallel code, portability, little overhead in terms of storage and in computation. The parallelization paradigm found to be most effective for our software on distributed memory machine was to provide the user with a “*client/server*” architecture.

3.1 The client/server architecture

Let us consider a parallel machine with P processors. Each processor has its own memory and processors communicate via message passing. According to the client/server architecture, we associate to each processor a process. The process which corresponds to the existing non-parallel code is called *client* and the *server* is composed by $P - 1$ processes implemented in *SPMD* model. The client initiates the conversation and is obtained by adding to the non-parallel code, some subroutines calls that perform: data structure for parallel computing, distribute and redistribute some data during the computation of $K_\sigma = LDL^T$, $y = Mx$ and $K_\sigma^{-1}x = b$ by the server.

The server is implemented in order to receive and compute some services. Among these services, it has to: perform local data structure for parallel computing for each processor; compute in parallel of the computations $K_\sigma = LDL^T$, $y = Mx$ and $K_\sigma^{-1}x = b$.

The most difficult part to design is the *SPMD* code which is initiated by a client process to speedup the computing time. To do this, we first remark that the algorithms associated to the three major computations consist on *nested loops*. Second, we reuse the principles of the *systolic* model for displaying of the potential parallelism inside the *nested loops* and justifying the aggregation of iteration so as to reduce communication overhead while exploiting coarse-grained parallelism. Each aggregate is a block of fine-grained computations located in different hyperplane of a given space. It also defines an atomic unit of computation i.e no synchronization or communication is necessary during the execution of the fine-grained computations inside a block. Thus, all necessary data must be available before such atomic executions. This imposes the constraints that splitting the set of fine-grained computations does not result in deadlocks.

Within this context, different optimization criteria have been proposed. Among these, *Tiling* (see [RS92]) is the most popular one. The main issues raised by tiling is the defining of *tiling* conditions and choosing an optimal *tile* size and shape. The most relevant results have been obtained with nested loops whose behavior is relatively predictable. Bearing this in mind, we propose a set of criteria which benefit from the combinatorial behavior of nested loops.

3.2 The data and computation partitioning

Following the methodology described by *Miranker*[MW84], *Moldovan*[Mol83], *Moreno*[ML88] and *Quinton*[Qui83], we use *uniform recurrence equations* to describe the sequential algorithm and model it into a directed graph, namely G . In this graph, D is the set of vertices and E the set of arcs. Each vertex $u \in D$ represents a fine-grained computation. A pair (u, v) in E means that the fine-grained computation at vertex v requires data from u . Note that in the graph G , vertices are locally connected according to the uniform recurrence equations model.

In figure 2, we have reported the systolic graph for the matrix-vector multiply $y = Mx$. For the matrix M , we only use its upper triangular part for the needs of the computation. In the diagram, we use the following representation: a circle is located at a point (i, i) and represents the computation associated to the diagonal term of M ($m_{i,i}$); a square is located at a point (i, j) , where $j > i$, and represents the computations associated to the extra-diagonal term $m_{i,j}$ of M . Dotted squares are only used for legibility. Such a model is also used in [BPST96].

In figure 4, we have reported the systolic graph for the lower triangular solve followed by the upper triangular solve. For these two computations, we only use the upper triangular part of the matrix K_σ which contains the result of the *Gaussian* factorization LDL^T . In the diagram, we use the following representation: a circle is located at a point (i, i) and represents the computation associated to the diagonal term of D ; a square is located at a point (i, j) , where $j > i$, and represents the computations associated to the extra-diagonal term of L^T .

In the figure 5, we have reported the systolic graph for the LDL^T factorization of the matrix K_σ . In the diagram, we use the following representation:

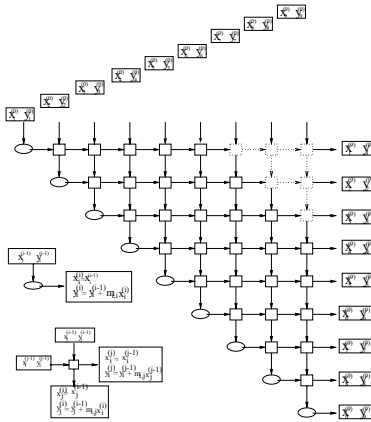


Fig. 2. The systolic graph for the matrix-vector multiply $y = Mx$.

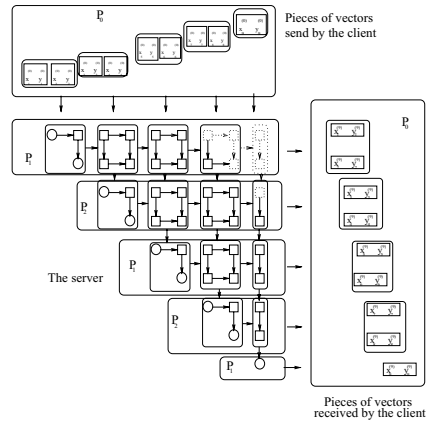


Fig. 3. The block splitting of the systolic graph.

a black circle, located at point (i, i, i) represents the computation of a needed value $d_{i,i}^{-1}$; a white circle, located at point (i, i, k) , for $i > k$, represents a partial modification of a diagonal term; a black square, located at point (i, j, i) , for $j > i$, represents the needed term in matrix L^T ; a white square, located at point (i, j, k) , for $i < j$ and $i, j > k$ represents partial modification of an extra-diagonal term. Dotted square and circle are only used for legibility.

Given a graph G as previously defined, we now describe the block aggregation strategy. Assuming that D , the set of nodes of G is a subset of N^q , where $q = 2, 3$, we “split” G into N blocks of iteration in the first dimension in N^q . The same split is applied for the remaining dimensions and thus a qD partitioning is obtained. In the figure 3, we have illustrated such splitting strategy for the matrix-vector multiply. We do not have enough space here to illustrate it for the two other operations.

Our aim is to “split” G into different blocks of nodes. Each of them will correspond to a single node in a new graph, say G' . Let u and v be two nodes in G' and $A(u, v)$ be the set of arcs linking the two blocks of nodes in G corresponding to u and v . If $A(u, v) \neq \emptyset$ then a pair (u, v) defines an arc in G' .

Finally, we propose a logic of allocating computations and data to available processors. According to uniform recurrence, a final result, say Y , is obtained after successive modifications of an input the piece of data, say $Y(u_0)$. Let $(Y(u_k))_{k \geq 0}$ be the sequence of these successive modifications. Note that u_k is a point of G' and $Y(u_k)$ is a value produced by the node u_k and required by u_{k+1} for its computations. We note that, for many numerical algorithms, piece of data Y is reused to store the terms $Y(u_k)$ during one or more iterations. Besides, in several cases we also notice that $\forall k \geq k_0$ terms $Y(u_k)$ may be localized along the same line. Then the idea of allocation strategy is: -to consider the intersection of D with a set of parallel hyperplanes containing such lines, -to al-

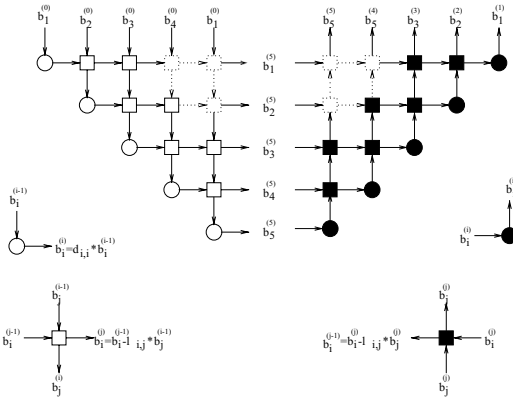


Fig. 4. The systolic graph for the lower and upper triangular system solve.

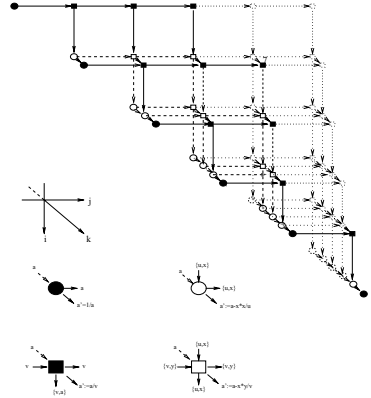


Fig. 5. The systolic graph for the LDL^T solve.

locate such lines to available processors under the constraint of the optimal time function. In the figure 3, we have illustrated a block allocation strategy for the matrix-vector multiply. The processors are represented by P_{num} , for $num = 0, 2$. The processor P_0 is the client and the processors P_1 and P_2 are the ones used for the server.

4 Experimental results

To get the potential performance of the software on distributed memory machines, some examples have been run on a workstation network and on the *Intel Paragon*. One of the most important class of application tested arise in structural analysis. Here, the matrices K and M are obtained after applying the finite element method to the *D'Alembert* equations. One example we have worked with during the experimentations involves a cylinder constructed from 3D finite elements. The model have up to 21,000 degrees of freedom.

4.1 Sequential performances

The sequential version of the software was written in Fortran and tested on a *Sun Sparc5* workstation. In figure 6, we summarize for two test problems, the main characteristics of the eigenvalue problem and the performance number of the serial software. The number of degrees of freedom is represented by n . The matrices K and M are both skylines and have the same storage management. The number lbw and nz are respectively the number of coefficients per line and in the upper triangular part.

Given a test problem, the goal is to compute few eigenvalues after l iteration of the *block Lanczos* method. Each computed eigenvalue has a multiplicity up to

	r	l	n	nz	lbw
CYL_1	4	23	21960	8169300	378
CYL_2	4	8	11550	5544141	642

	<i>The performance in seconds</i>				
	$K_\sigma = LDL^T$	$y = Mx$	$K_\sigma^{-1}x = b$	<i>other</i>	<i>total</i>
CYL_1	1505.2	12475.0	2619.5	2919.5	19519.3
CYL_2	1300.9	2550.9	288.1	219.9	4359.8

Fig. 6. The main characteristics for two test problems and the performance numbers on a sparc station.

r . In the table of the figure 6, we have summarize the performance number of the sequential software. According to these results, one observes that the majority of the serial computation occurs in the three operations: $K_\sigma = LDL^T$, $y = Mx$ and $K_\sigma^{-1}x = b$.

4.2 Parallel performances

For the implementation of the *client/server* architecture, we use the *BLACS* primitives to allow the communication between processors. Little consideration is given to the physical topology.

The first result reported were obtained on a network of workstations. Such a collection of serial computers appears as one large virtual parallel machine in which the processors are connected by *Ethernet*. Each processor is a *Sun Sparc5* workstation and the *Ethernet* is a *10Mbits/s* broadcast bus technology with distributed access control. The table of the figure 7 summarizes the performance number on such virtual parallel machine. The tested problem is CYL_1 . The best performance numbers were observed with $P = 8$. A comparison between the performance numbers of the tables in figure 6 and 7 shows that the parallelization reduces the execution time for the three operations $K_\sigma = LDL^T$, $y = Mx$ and $K_\sigma^{-1}x = b$ and thus for the global execution.

	<i>partitioning</i>	$K_\sigma = LDL^T$	$y = Mx$	$K_\sigma^{-1}x = b$	<i>other</i>	<i>total</i>
$P = 8$	728.4	362.3	1557.8	729.4	2113.4	5491.3
$P = 11$	921.6	653.1	2248.7	1147.7	2803.3	7824.4
$P = 13$	1095.0	602.6	2211.4	1102.8	2761.9	7773.7
$P = 16$	1651.0	452.5	2264.8	1152.9	2809.9	8331.1

Fig. 7. Test problem CYL_1 : the performance number for distributed memory software.

Finally, we report some performance numbers concerning the execution time for the same operations aforementioned on the *Intel Paragon*. The tested problem is CYL_2 . Each operation, for $P > 1$ is executed 50 times and we have

reported the medium value. One more time, the parallelization allows significant reduction of the execution time. However, as the size of the matrice increases, the partitioning (not reported for the *Paragon*), leads to some overheads.

	$K_{\sigma} = LDL^T$		$y = Mx$	$K_{\sigma}^{-1}x = b$
$P = 1$	843.55	$P = 1$	5.36	6.52
$P = 13$	74.96	$P = 8$	0.39	0.42
$P = 16$	69.80	$P = 10$	0.42	0.41
$P = 20$	54.69	$P = 12$	0.45	0.42

Fig. 8. Test problem CYL_2 : the performance numbers on the Intel Paragon.

5 Conclusion

In this paper, we have presented an explicit parallelization strategy for porting existing sequential application on distributed memory machine. The idea of our strategy is to reuse the *systolic algorithm* principle for the display of both distributed data structures and the programming model.

The distributed software presented in this report is suitable for parallel *MIMD* computers and network of workstations. However, due to the bottlenecks imposed by the partitioning step and to some non-parallel operation which are executed by the “client”, we could not test the problem with more degrees of freedom. To treat such problems, we are seeking to eliminate these bottlenecks by pursuing research into using the *systolic algorithm* principle for the parallelization of full orthogonalization operations and the parallelization of the partitioning step.

References

- [BPST96] J-C. Bermond, C. Peyrat, I. Sakho, and M. Thuenté. Parallelization of the gaussian elimination algorithm on systolic arrays. *Journal of parallel and distributed computing*, 33:69–75, 1996.
- [Cha88] F. Chatelin. *Valeurs Propres de Matrices*. Masson, 1988.
- [ML88] J.H Moreno and T. Lang. Graph-based partitioning for matrix algorithms for systolic arrays: Application to transitive closure. *ICPP*, 1:28–31, 1988.
- [Mol83] D.I Moldovan. On the design of algorithms for vlsi systolic arrays. *Proc.IEEE*, 71:113–120, 1983.
- [MW84] W. L. Miranker and A. Winkler. Space-time representation of computational structures. *Computing*, 32:93–114, 1984.
- [Qui83] P. Quinton. The systematic design of systolic arrays. Technical Report 193, IRISA, 1983.
- [RS92] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, October 1992.

Heading for an Asynchronous Parallel Ocean Model

Josef Schüle

Institute for Scientific Computing, Technical University Braunschweig, D-38092
Braunschweig, Germany, phone: +49 531 3915542, fax: +49 531 3915549, email:
`j.schuele@tu-bs.de`

1 Introduction

The Swedish Meteorological and Hydrological Institute (SMHI) makes daily forecasts of currents, temperature, salinity, water level, and ice conditions in the Baltic Sea. These forecasts are based on data from a High Resolution Operational Model of the Baltic Sea (HiROMB) currently calculated on one processor CRAY C90 vector computer and has to be ported to the distributed memory parallel CRAY T3E to save operation expenses and even more important allow a refinement of the grid resolution from currently 3 nautical miles (nm) to 1 nm within acceptable execution times.

The 3 nm grid covers the waters east of 6°E and includes the Skagerrak, Kattegat, Belt Sea and Baltic Sea. Boundary values at the open western border for flux, temperature, salinity, and ice properties are provided by a coarser 12 nm grid covering the whole North Sea and Baltic Sea region.

In a rectangular block covering the 3 nm grid only 25% of the surface points and less than 10% of the volume points are active, t.m. water points which have to be considered in the calculation. To avoid indirect addressing, the grid is decomposed into a set of smaller rectangular blocks discarding all blocks without water points. This reduces the fraction of inactive points significantly to the expense that one layer of ghost points is added around each block. The remaining blocks may be assigned to processors in a parallel environment. Introduction of a minimal depth for each block further reduces the number of inactive points.

Three different parts may be identified in the ocean model. In the **baroclinic part** water temperature and salinity are calculated for the whole sea including all depth levels. Explicit two-level time-stepping is used for horizontal diffusion and advection. Vertical exchange of momentum, salinity, and temperature are computed implicitly.

A semi-implicit scheme is used in the **barotropic part** for the vertically integrated flow, resulting in a system of linear equations (the Helmholtz equations) over the whole surface for water level changes. This system is sparse and non-symmetric, reflecting the 9-point stencil used to discretize the differential equations over the water surface. It is factorized with a direct solver once at the start of the simulation and then solved for a new right-hand side in each time step.

Ice dynamics includes ice formation and melting, changes in ice thickness and compactness, and is taking place on a very slow time scale. The equations are highly nonlinear and are solved with Newton iterations using a sequence of linearizations. A new equation system is factorized and solved in each iteration using a direct sparse solver. Convergence of the nonlinear system is achieved after at most a dozen iterations. The linear equation systems are typically small. In the mid winter season however, the time spent in ice dynamics calculations may dominate the whole computation time.

In the original, serial version of HiROMB, the linear systems of equations for the water level and the ice dynamics were solved with a direct solver from the Yale Sparse Matrix Package (YSMP) [1]. For parallel HiROMB we were urged to introduce as few modifications into the Fortran 77 code as possible and therefore employ a distributed multi-frontal solver written by Bruce Herndon [2].

In the multi-frontal method processors factorize their own local portions of the matrix independently and then cooperate with each other to factorize the shared portions of the matrix. In Herndons' solver, these shared equations are arranged hierarchically into an elimination tree, there the number of participating processors is halved when advancing in the hierarchy. Thus, if the fraction of shared equations is large, performance will suffer due to lack of parallelism. As an additional drawback, the solver requires a power-of-2 decomposition of the matrix.

Parallelization is introduced with the Message Passing Interface (MPI) library [3]. More details on the reorganization of the code are given elsewhere [4, 5].

2 Decoupling of the Computational Domains

As mentioned above, HiROMB consists of two major parts. One is responsible for the water properties, which itself consists of the barotropic and the baroclinic part, and one part is responsible for the ice dynamics. These parts are tightly intertwined. Water temperature and salinity are interdependent with ice formation or melting as are water flux and ice drift. Specific dependencies of ice properties occur at river inlets and grid boundaries.

These dependencies exist within each grid. The different grids are loosely coupled at the western edge of the finer grid.

Using the same decomposition for water and ice calculations, the decomposition during winters will have to be a compromise with unavoidable load imbalances. In midwinter time, the execution time may be six times as large as during summer. At the same time, the relative smallness of the ice covered region prevents an effective parallel execution (see Figure 2), because the overhead introduced by context switching between many small-sized blocks overwhelms the parallel benefit.

But even in summer time, the decomposition is a compromise between the baroclinic and the barotropic part. These parts are strongly coupled what pre-

vents the usage of different decompositions. Because the coarsest grid with 12nm may not execute on more than 8 PEs effectively, 65 processors (PEs) currently execute only 4.3 times faster than 5 PEs do. An example of the 3nm grid decomposed into 24 blocks on 16 PEs is shown in Figure 1.

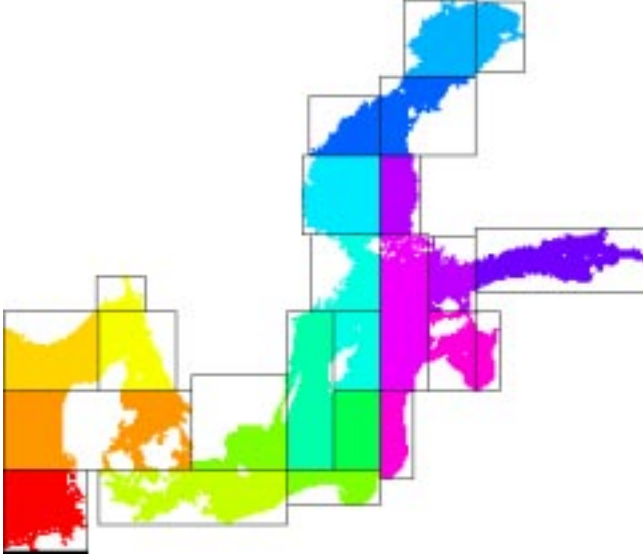


Figure 1. This is a decomposition of the 3nm grid into 24 blocks distributed onto 16 color coded processors.

To overcome these deficiencies, we intend to run HiROMB as a set of independent modules for the ice part and the water part in each grid. These modules then may be puzzled together on any number of PEs to result in an asynchronous time step with shortest possible execution time. This will be achieved in a number of steps:

1. The computation for each module is grouped together and separated.
2. An independent decomposition of the water and ice part is introduced.
3. The communication pattern of each module is organized to allow overlapping without conflicts.
4. An asynchronous execution order of the modules, that is a less coupled system, is investigated to reduce the execution time within mathematical error bounds.

Point 1 requires only minor changes to the code except for the ice thermodynamics and its influence on the water temperature. This is resolved by passing over the correction of the water temperature to the next time step.

Due to the design of the data structure (cf. [4, 5]), the specification of an own independent static decomposition for the ice part is simple. Only some additional

routines have to be introduced to handle data exchange between water blocks and ice blocks. Thereby care has been taken to avoid MPI calls there water blocks and ice blocks are fully or partly located on the same PE and a simple memory-to-memory copy is performed instead.

But the ice domain is subject to dynamic changes. Ice may melt and water may freeze during a 24h forecast run. Melting simply reduces the number of active grid points in the ice decomposition and therefore may reduce the effectivity of a decomposition slightly. Modifying a decomposition because of ice melting is expected to introduce a significant communication overhead, that we estimate to exceed its benefits.

Freezing of water may occur at the edge of an ice flake, but likewise new flakes may be formed (compare to Figure 2). Opposite to melting, freezing will not only influence the effectivity of a decomposition, but its usability in cases there new ice is formed outside the covered domain. Therefore all water points have to be inspected in each time step to locate new ice to decide whether or not a new decomposition is required. Because the growth of existing ice flakes is most probable, a halo region is introduced around each flake in the first decomposition. The size of the halo region is made dependent on the water temperature. Currently we neither know the costs for introducing a new decomposition nor the costs for introducing inactive points in this halo region.

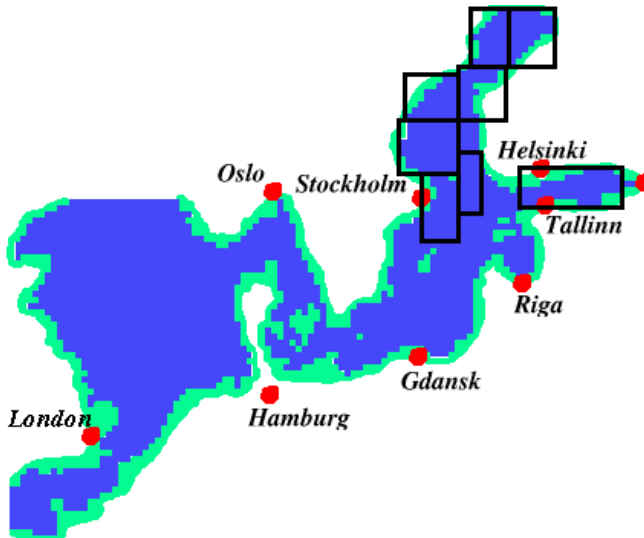


Figure 2. This is a decomposition of the ice covered region in the 12nm grid into 8 blocks distributed onto 8 processors. Only small regions in the northern and eastern part of the Baltic Sea are usually covered with ice.

Point 3 requires the introduction of distinct MPI communicators for each module. Still a master PE is required for each module to handle I/O requests and to organize the update of boundary values within each module and data exchange between modules. Currently we want to avoid introducing separate masters for each module. Whether this single threaded master becomes a bottleneck or not is still under investigation. In case it will, the calculation of boundary values and the interpolation and extrapolation between the grids will be handed over to working PEs. As a second possibility to increase performance on the master, blocking communications in the update routines may be exchanged against nonblocking ones. This replaces the code-driven execution order currently used against a demand-driven execution order on the master and requires a fair amount of recoding. Waiting times of slave PEs are currently analysed in details.

3 Results

Timings for an artificial 4 hours simulation on a total of 9 PEs of a CRAY T3E-900 are given in table 1. One of the PEs is required as master PE. A parallel run with a common decomposition for water and ice employing 4 PEs on the coarse 12nm grid and 8 PEs on the finer 3nm grid requires 811 sec. This has to be compared with 959 sec for the same number of PEs per grid but with two different decompositions for the water and ice parts. Obvious, the introduction of a second decomposition introduces a lot of overhead. However, decreasing the number of PEs employed for the ice dynamics pays out. In the fastest version requiring 548 sec, i.e. only 70% of the original time, the ice dynamics for the coarser 12nm grid are calculated serially, while the finer 3nm grid is distributed onto 2 PEs.

12nm Grid		3nm Grid		
Water	Ice	Water	Ice	Time (sec.)
Common Decomposition				
4	4	8	8	811
Different Decompositions				
4	4	8	8	959
4	4	8	4	789
4	2	8	4	570
4	2	8	2	558
8	2	8	2	618
4	1	8	2	548

Table 1. These timings were done for a 4 hour simulation with data output every 3 hours using a CRAY T3E-900. In all calculations 9 PEs were employed. Ice growth was pushed artificial to enforce severeral different ice decompositions during the run.

4 Conclusion

Moving the production of HiROMB forecasts from a vector processor to the massively parallel computer CRAY-T3E is feasible but results in poor speedups prohibiting the usage of more than 17 PEs in production. To overcome this deficiency load balancing has to be improved and overhead introduced by fine grain parallelization has to be decreased.

As a first step, two different domain decompositions for ice and water calculations in each grid are introduced. This fastens up a calculation with 16 PEs on a CRAY T3E-900 considerably.

As next step, we are heading towards a completely asynchronous version, there the water calculations and the ice dynamics in each grid are treated as independent modules. This will help to speed up the program further.

References

1. S. C. Eisenstat, H. C. Elman, M. H. Schultz, and A. H. Sherman. The (new) Yale sparse matrix package. In G. Birkhoff and A. Schoenstadt, editors, *Elliptic Problem Solvers II*, pages 45–52. Academic Press, 1994.
2. Bruce P. Herndon. *A Methodology for the Parallelization of PDE Solvers: Application to Semiconductor Device Physics*. PhD thesis, Sanford University, January 1996.
3. Mark Snir, Steve Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. MIT Press, Cambridge, Massachusetts, 1996. ISBN 0-262-69184-1.
4. T. Wilhelmsson and J. Schüle. Fortran memory management for parallelizing an operational ocean model. In F. Hertweck H. Lederer, editor, *Proceedings of the Fourth European SGI/Cray MPP Workshop*, pages 115–123, Max-Planck-Institut, Garching, 1998. IPP R/46.
5. J. Schüle and T. Wilhelmsson. Parallelizing a high resolution operational ocean model. In A. Hoekstra B. Hertzberger P. Sloot, M. Bubak, editor, *High-Performance Computing and Networking*, pages 120–129, Berlin, Heidelberg, 1999. Springer.

Distributed Collision Handling for Particle-Based Simulation

Giancarlo Frugoli¹, Alessandro Fava¹, Emanuele Fava¹, and Gianni Conte²

¹ Industrial Engineering Department, University of Parma,
Parco Area delle Scienze 181/A, I-43100 Parma, Italy
`frugoli@ied.unipr.it`
`http://kaemart.unipr.it`

² Computer Engineering Department, University of Parma,
Parco Area delle Scienze 181/A, I-43100 Parma, Italy
`conte@ce.unipr.it`

Abstract. The work is concerned with the parallel implementation of a simulator of the dynamics of non-rigid materials. In particular, it has been realized a distributed version of the algorithm that manages self-collisions and collisions among different deformable objects, as this results one of the most time-consuming parts of the original program. The implementation has been done using the MPI message-passing library on a cluster of low-cost personal computers running Linux as operating system. Some preliminary tests porting the parallel code as-is on a Silicon Graphics Onyx2 shared-memory system has been also carried out.

1 Introduction

Computer simulation systems are today commonly used in several industrial sectors during the design and the test phase of a product. Existing CAD and off-line programming tools have been proved to be effective in designing and evaluating manufacturing equipment and processes, but they cannot be used to study and simulate processes involving non-rigid products. In fact they do not take into account material deformation due to environmental factors such as gravity, wind, collisions with obstacles. In general, it is not possible to predict the dynamic behavior of the product when it is manipulated.

Modeling and simulation of flexible products is therefore attracting more and more people from research and industrial communities. Techniques and tools adopting a physically-based approach have been developed and experimented with success. One of these systems is the particle-based simulator named Soft-World [1], developed by the Industrial Engineering Department of the University of Parma.

One of the major problems of dynamic simulation is the computation speed, especially when dealing with a high number of potential collisions and constraints: complex and detailed simulations can easily take several hours of computation on standard workstations. The computational cost of simulating a flexible object depends on the level of detail of the model, but also on other variables,

such as the material stiffness, the constraints the model has to respect and the number of collisions among objects that occur during the simulation.

Despite the continuous computer performance growth, the computational power remains one of the most important limiting factors in the field of simulation of the dynamics of flexible bodies. Therefore it is important to experiment the use of high-performance parallel and distributed architectures.

In this context it has been implemented a parallel version of *SoftWorld* [2]. The new version has been developed and tested on a distributed architecture named *ParMa2* (Parallel Machine 2) [3], based on four Dual Pentium II personal computers running Linux as operating system and connected via a high-speed 100 Mbit/s network. *SoftWorld* is written in C language and the distributed version is based on *MPICH*, a freely available implementation of the MPI standard message-passing libraries [4]. The results obtained show an almost linear speed-up with the number of processors, allowing to reduce the computation time or to obtain, with the same elaboration time, longer and more detailed simulations.

The tests on the PC cluster have shown the efficiency of this low-cost architecture when dealing with complex problems, maintaining an excellent cost/performance ratio, especially with respect to commercial multi-processor architectures. It has been therefore demonstrated that with this approach it is possible to deal with highly complex problems with reasonable costs.

2 Non-rigid Material Simulation with *SoftWorld*

With a physically-based approach, an object is modeled using a mathematical representation that incorporates forces, torques, energies and other attributes of Newtonian mechanics. Such a model reacts in a natural way to external forces and constraints applied.

The particular physically-based model used within our prototype is the particle-based model [5], also known as the mass-spring model, which represent non-rigid objects by sets of particles characterized by physical properties. The interaction laws among the particles are modeled by means of forces that determine the dynamic behavior of the material. The particle-based model can be considered a macro-molecular description of the material, where artificial forces acting among the particles are introduced to simulate the inter-molecular attraction and repulsion forces. They describe the way the material reacts to external influences and determine parameters such as elasticity, viscosity and plasticity.

The particle system is governed by a mathematical model composed of a system of second order ordinary differential equations, that can be solved step by step with the traditional numerical integration algorithms.

3 Analysis of the Original Source Code

The first step to obtain a parallel version of a sequential code is to find possible approaches to the problem, analyzing which part of the elaboration process can

be distributed on more processors. It is moreover fundamental that the elaboration results are coherent with those obtained from the original program and, of course, that there is a real performance improvement.

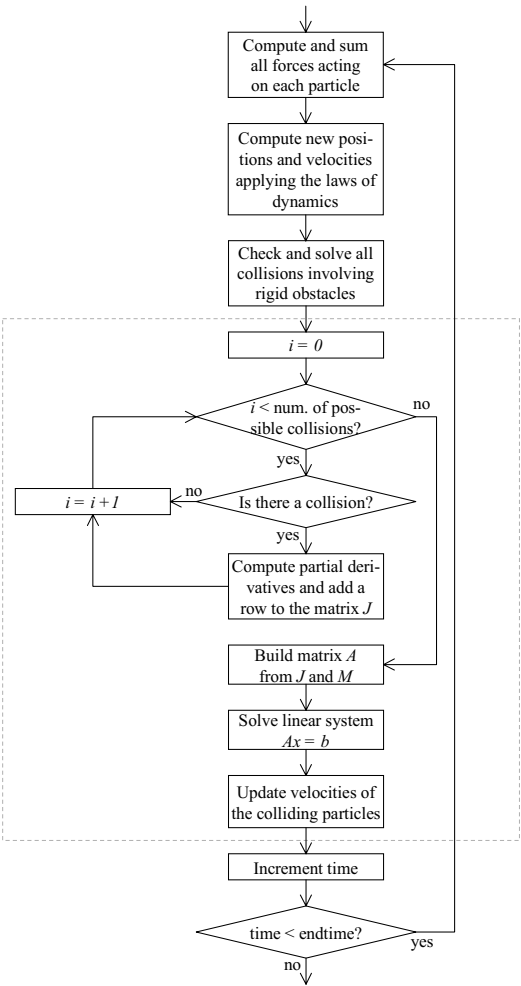


Fig. 1. The simulation algorithm used in the sequential version of the simulator: the dotted line encloses the detailed description of the management of collisions involving only non-rigid parts

The algorithm used by our simulator consists mainly of a loop in which the state of the system (i.e., positions and velocities of all the particles) is cyclically updated to reflect the new dynamic situation after a fixed time step, until the simulation time reaches the value specified by the user (see Fig. 1).

The analysis of the elaboration times of the single functions highlighted that in most cases more than 90% of the total time is spent during the collision handling phase, whose computational complexity is generally greater than $O(n^2)$ where n is the number of particles.

Regarding the collisions of particles with rigid obstacles, our simulator uses a simplified technique that corrects the velocities of some particles immediately after having detected a collision; then continues the search starting from this altered state. This intrinsic sequential structure makes impossible to parallelize this function.

A different technique is used for managing collisions that involve only non-rigid parts (i.e., self-collisions and collisions among different flexible objects). In this case SoftWorld uses inequality constraints [6]. This method consists in the numerical solution of a linear system of type $Ax = b$ representing the conservation of momentum for the particles involved in the collision.

The geometry used for collision checking is that of polyhedra with triangular faces and particles at every vertex. The search for collisions is done by checking vertices vs. triangles as well as sides vs. sides.

The matrix A is obtained from matrix J , containing the partial derivatives of the constraints applied (including the non-penetration constraints that implement collisions handling), and from matrix M , containing the masses of the particles involved. The matrix J is built step by step by computing partial derivatives for each particle/triangle pair and each side/side pair found in colliding contact during the collision checking phase. In the current version of the system, the set of pairs to be checked for collision is a data given as input, therefore the maximum number of simultaneous collisions is fixed and known since the beginning of the simulation.

Finally, by solving the linear system, we can find the instantaneous jump in velocity that make particles react realistically to the collision.

4 Parallel Implementation

The approach adopted consists in maintaining the original structure of the sequential code, except during the collision handling phase, which will be distributed among the available nodes.

The parallel version of the simulator consists of a single executable program running on all the nodes. While performing the parallel section, each processor elaborates simultaneously its part of collision management. On the other parts of the program each node replicates the original sequential elaboration. The parallel code has been developed by partitioning the loop that checks for collisions and distributing it among n processes (processors).

There is a *master* process with the responsibility of synchronizing the other processes by collecting their partial results and broadcasting the new state to be used during the next iteration (see Fig. 2). At each iteration, each *slave* process i communicates to the *master* the number of actual collisions found during the current time step, along with the corresponding J_i submatrix. The *master* will

merge these data in order to reconstruct the whole matrix J containing all the necessary values to calculate the collision response.

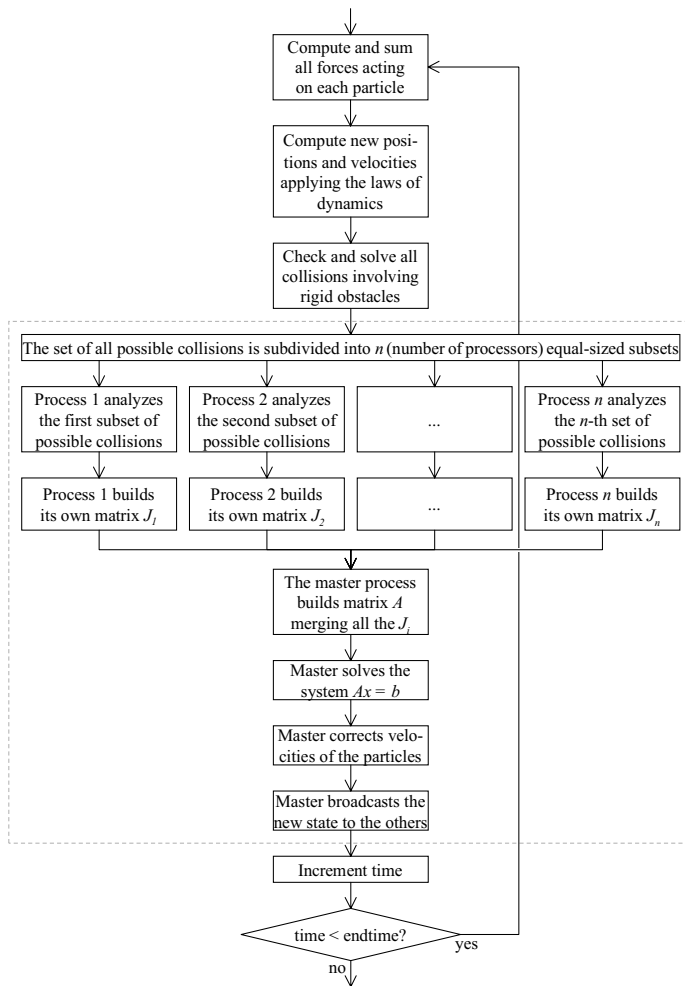


Fig. 2. Parallel version of the simulation algorithm: the loop checking for collisions involving only non-rigid parts has been distributed among n parallel processes

5 Results

One of the test examples used to evaluate the performance improvement of the distributed version concerns the simulation of a ribbon falling and crumpling on a rigid plane (see Fig. 3). This example is particularly heavy from the point

of view of collision handling. In particular, more than 90% of the collisions that occur are self-collisions and only the remaining portion involves the rigid obstacle that characterizes the support plane. These values point out that the sequential execution of the collision checking with rigid obstacles does not alter the effect of the parallelization of the other functions.



Fig. 3. Simulation of a ribbon falling and colliding with a table as well as with itself

The example has been tested on a cluster composed of four Dual Pentium II 450 MHz PCs connected by a 100 Mbit/s switched Fast Ethernet network, forming an 8-processor system. The results obtained running the same simulation with different numbers of processors are presented in Table 1.

Table 1. Results obtained for the falling ribbon simulation

Number of nodes	1	2	3	4	5	6	7	8
Total time (s)	3134	1707	1259	964	792	680	600	540
Collision handling (s)	3008	1535	1027	777	623	522	452	399
Communication (s)	0	136	196	151	134	122	113	106
Speed-up	1	1.84	2.49	3.25	3.96	4.61	5.22	5.8

These results are encouraging, as we reached a performance of nearly six times that of the sequential version when using all the eight processors available. The speed-up seems to be almost linear with the number of nodes used, as shown in

Fig. 4, thus promising potential increase of performance on a cluster with more processors.

Preliminary tests porting the code as-is on an 8-processor Silicon Graphics Onyx2 shared-memory system based on a NUMA (Non-Uniform Memory Access) architecture, have shown that the speed-up is comparable with that of the PC cluster. However, the performance obtained with the SGI system does not seem to be consistent with the potential of the architecture. It is therefore necessary an in-depth study of the problem for this particular architecture, for which probably a message-passing approach does not represent the best choice.

Anyway, these preliminary results confirm the excellent cost/performance ratio of clusters of personal computers such as the ParMa2 system, which prove to be attractive low-cost high-performance computing platforms.

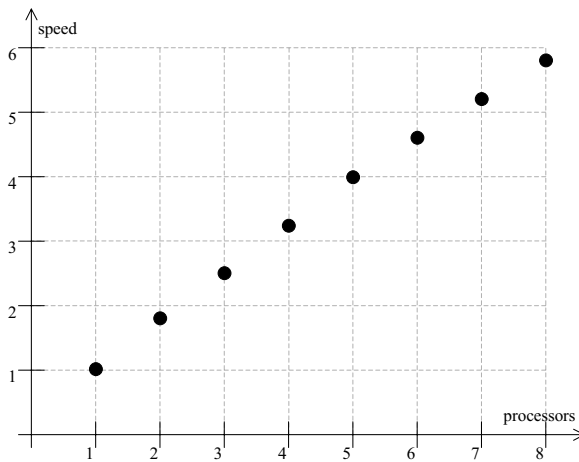


Fig. 4. Speed-up obtained using different numbers of processors with the same example

6 Conclusions

This research work aimed at implementing a distributed version of SoftWorld, a simulator of the dynamics of non-rigid objects developed by the Industrial Engineering Department of the University of Parma.

After an analysis of the source code, the collision checking and response phase has been individuated as the most time-consuming part of the program. The parallelization work has been concentrated in particular on the handling of self-collisions as well as other collisions involving only non-rigid parts.

For the development of the parallel version, it has been used the MPI message-passing library, which can exploit the potential of low-cost architectures

such as clusters of personal computers. The use of the MPI standard allows furthermore to obtain code easily portable on different architectures and operating systems.

The parallel version has been tested on an 8-processor PC cluster running Linux and some preliminary tests have been done porting the code as-is on a Silicon Graphics Onyx2 shared-memory system. The excellent results obtained on the cluster of personal computers, demonstrate the efficiency and the attractive cost/performance ratio with respect to commercial multi-processor architectures.

References

1. Frugoli, G., Rizzi, C.: Physically-Based Simulation with SoftWorld. In: Proceedings of the International Conference on Applied Modelling and Simulation. Honolulu, Hawaii (1998) 373–377
2. Fava, A.: Realizzazione in Ambiente Distribuito di un Simulatore del Comportamento di Oggetti Non Rigidi. Master's Thesis in Electronic Engineering. University of Parma, Italy (1999)
3. Vignali, D.: ParMa2 White Paper. Department of Computer Engineering, University of Parma, Italy (1998). Available at: <ftp://ftp.ce.unipr.it/pub/ParMa2>
4. Groop, W., Lusk, E.: User's Guide for MPICH, a Portable Implementation of MPI. Mathematics and Computer Science Division, University of Chicago (1995)
5. Witkin, A.: Particle System Dynamics. In: An Introduction to Physically Based Modeling. SIGGRAPH Course Notes, Vol. 34 (1995)
6. Platt, J.: Constraint Methods for Neural Networks and Computer Graphics. Ph.D. Thesis in Computer Science. California Institute of Technology (1989)

Parallel watershed algorithm on images from cranial CT-scans using PVM and MPI on a distributed memory system

Cristina Niclescu, Bas Albers and Pieter Jonker

Delft University of Technology
Faculty of Applied Physics
Pattern Recognition Group
Lorentzweg 1, 2628CJ Delft, The Netherlands
email:cristina,bas,pieter@ph.tn.tudelft.nl

Abstract. In this paper we present a parallel implementation of an image segmentation transform - the watershed algorithm. The algorithm is implemented using the message passing paradigm, with both PVM and MPI and is adapted to a specific application of determining the volume of liquor from CT-scans. Our watershed algorithm is applied on 2D images from cranial CT-scans to extract the liquor parts from them. The evaluation on a distributed memory system is included.

1 Introduction

Image segmentation is important in image processing. Image segmentation is often one of the first steps in applications like object tracking, image compression or image enhancement. A wide variety of algorithms are used for image segmentation.

One of the good algorithms used to segment an image is the watershed filter. The algorithm can be applied to find the ridges and crests in an image. The algorithm is very precise but it has the disadvantage that it produces oversegmentation. Several watershed algorithms were developed and the main goal is how to reduce the oversegmentation. We have implemented a parallel watershed algorithm for the HPISIS project[7]. The project is aimed at the development of a system that can monitor atrophic and traumatic brain changes by means of high speed automated analysis of the brain/liquor ratio from cranial CT-scans. Our approach is to first segment a CT-scan (a 3D image) and then calculate the brain and liquor volume. A CT-scan of the human brain is a series (typically 20-30) of 2D images. The 2D images are consecutive slices of the brain, so the entire set can be seen as a 3D image. Because the sampling frequency in the z-direction (vertical when the patient is standing) is very poor, we used an interpolation method to obtain even sampling. To determine the brain/liquor ratio we must calculate the volume of each liquor enclosure in the brain. Alternatively, from each 2D slice the liquor/brain ratio can be calculated and accumulated to a total amount, but the statistics on the volume blocks will be lost. Moreover, in areas around nose and ears the allocation of areas to volume blocks is difficult

if the 3D connectedness is neglected. We have used the watershed algorithm to find the liquor areas from a CT-image.

We implement first sequential 2D and 3D variants of the watershed algorithm to obtain a ground truth for our speed-up figures. Determination of the brain/liquor ratio considering a CT scan of 20 slices as a set of 2D images, takes about 5 minutes on a Sparc20 and about 15 minutes if processed as a 3D image. To investigate speed-up by parallelization we implement and test a parallel 2D watershed algorithm.

This paper is organized as follows. Next section presents related work. Section 3 describes the proposed parallel watershed algorithm. Speed-ups obtained on a distributed memory system with 24 nodes for different image sizes are graphically presented and interpreted in Section 4. Finally, Section 5 concludes the paper.

2 Related work

There are several methods to segment an image using a sequential watershed algorithm[1, 2]. In the algorithm of Vincent and Soille[1] all the pixels from the image are sorted in the order of increasing greylevel. The pixels with the lowest greylevel are taken as seeds for the basins and from that basins regions which include the pixels with the next greyvalue are grown in a breath-first manner. If the pixels from a propagation front (basin) do not touch pixels from another propagation front, they are assigned the same label. Otherwise, the pixels from the propagation front are labeled as watershed pixels. All the pixels from a greylevel that cannot be reached in this way are new local minima and become seeds for new basins and the process is repeated till all the pixels from the image have been assigned a label.

Recently, parallel versions of the watershed algorithm have been also proposed [3, 4, 5, 6]. In the algorithm of Moga, Cramariuc, and Gabbouj[3] every node starts to compute the watershed of the sub-image assigned to it. A lot of communication is involved between nodes that process neighbour parts of the image. The algorithm of Meijster and Roerdink[6] consists of transforming the image in a directed valued graph $f^* = (\mathcal{V}, \mathcal{E})$. The vertices of the graph are maximal connected sets of pixels with the same greyvalue. If $v, w \in \mathcal{V}$ then $(v, w) \in \mathcal{E}$ if v contains a pixel p that touches a pixel q in w and $f(p) < f(q)$, where $f(p)$ represents the greyvalue of pixel p . Then, the watershed of the graph is calculated in a way similar to the sequential algorithm[1]. Finally, the graph is transformed back to an image. Since all the pixels having the same greyvalue belongs to one node of the graph the watershed calculation of the graph could be parallelized.

3 The proposed parallel watershed algorithm

The watershed algorithm is a region growing algorithm. There are several methods to parallelize the region growing algorithms on a distributed memory system[5].

The methods can be classified as:

- Sub-image parallel: The image is divided into sub-images which are distributed over the processors. Each processor processes the sub-image assigned to it. The processors have to exchange some data.
- Region parallel: The complete image is distributed to all processors. Each processor starts to grow regions from the seeds that are assigned to it. Since information assigned to one processor might be important to another, some communication between processors could be expected.
- Pixel parallel: All processors are assigned a disjunct number of pixels. The mapping of pixels to processors can vary from one algorithm to another. A lot of communication can be expected from this approach.
- Region and pixel parallel: In this method the region and pixel parallel are combined. A subset of processors is assigned to grow a certain region in a pixel parallel manner.

The key problem in the parallelization of the watershed algorithm is how to distribute the image over processors to achieve low communication time and high speed. We use a variant of the algorithm of Vincent and Soille[1] implemented first sequentially and then in a region parallel way. The entire image is distributed (in a multicast) over the n processors. In this way each processor has all the data needed to compute it's own result (i.e. a part of the output image), so the communication time is minimized. Each node starts computing the watershed in a certain region assigned to it and then the computed image is sent back to the master processor which combines the results. Because the watershed algorithm is a regional filter it may be clear that to compute a certain part of the output image a node needs more than the information inside the corresponding part of the input image assigned to that node.

The main problem is how to find the overlap between the regions of the input image assigned to different processors in order to compute correctly the watershed of the image. An example is presented in Figure 1. If the two local minima from the lower part of the image are not partially included in the overlap with the upper part of the image, the dotted watershed will not be found and the watershed of the image will not be computed correctly. So, the overlap around a part of the input image assigned to a processor has to be big enough to contain all the seeds of the regions of image assigned for computing to neighbour processors. All the pixels that can be reached from the edge of that part of the image in a non-descending way are part of the overlap. We use this observation in our algorithm.

4 Experimental results and discussion

Results of the parallel implementation of the watershed algorithm using PVM and MPI are presented in this section. The behaviour of the algorithm is studied on images of sizes 128x128, 256x256, 512x512 and 1024x1024. For MPI we

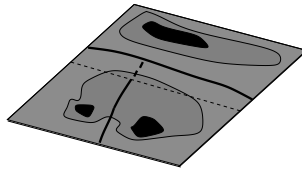


Fig. 1. Determination of overlapping regions.

also use 2 more images of sizes 257x257 and 384x384. An example of a test image is shown in Figure 2. As a preprocessing step a gradient magnitude filter with $\sigma = 1.5$ is first applied on the image. The result of applying the watershed algorithm on the pre-processed image is shown in Figure 3. The severe oversegmentation is clearly visible. To reduce the oversegmentation we use the histogram of the CT-image. This has 3 large peaks. From light to dark these peaks come from the background, the brain and the skull. To extract the skull we use a value derived from the histogram. Since CT-systems are calibrated, the value of the liquor has a predefined value that is used to extract the liquor regions. To extract the liquor parts, the basins with deepest greyvalues between

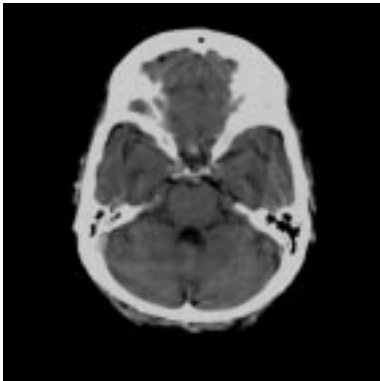


Fig. 2. Slice from a CT-scan.

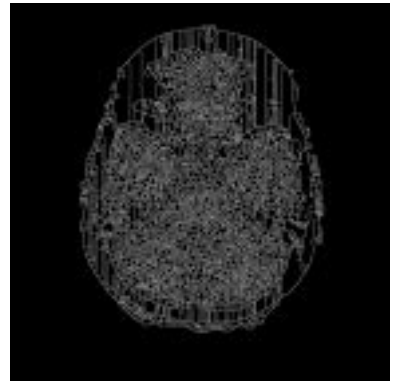


Fig. 3. Watershed of the image depicted in Figure 2.

0 and 40 are joined. The result is presented in Figure 4. The image from Figure 5 shows the result of extracting the skull. This is done by joining basins with lowest values between 170 and 255. It is quite easy to obtain the liquor areas inside the brain by combining the skull and liquor segmentation results. Figure 6 shows the final segmentation result. To compute the brain and liquor areas we count the number of pixels inside these regions as a measure of the surface. Figure 7

shows the skull segmentation in the 3D variant of the algorithm. The parallel

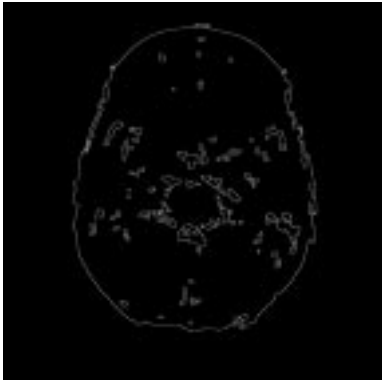


Fig. 4. Basin-joined method for liquor extraction from the watershed depicted in Figure 3.



Fig. 5. Basin-joined method for skull extraction from the watershed depicted in Figure 3.

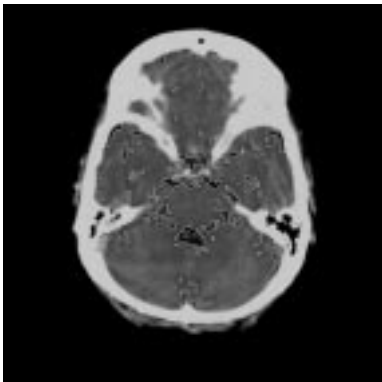


Fig. 6. Segmentation of the liquor regions inside the brain.

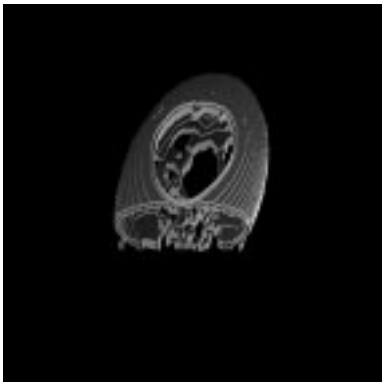


Fig. 7. Example of a 3D watershed extracting the skull volume

2D watershed algorithm is implemented using the message passing paradigm on a distributed memory system with 24 nodes. Both PVM and MPI are used to show the speed-ups and efficiency in each case. In Figures 8 and 10 the speed-up and efficiency of the parallel implementation of the 2D watershed algorithm using PVM are depicted for different image sizes. In Figures 9 and 11 the speed-up

and efficiency of the parallel implementation using MPI are depicted. The speed-up and efficiency of the algorithm when using PVM are not very good. This is because on the distributed memory system used for tests, PVM is implemented on top of TCP/IP and fast Ethernet. With MPI-Panda on top of Myrinet we obtained better results, largely due to the fact that Myrinet is faster (2Gb/s) and is based on wormhole routing crossbar switches. The large difference between the speed-up for 256x256 image size compared to 257x257 image size is due to MPI-panda implementation. When more than 64kB are sent, another protocol is used.

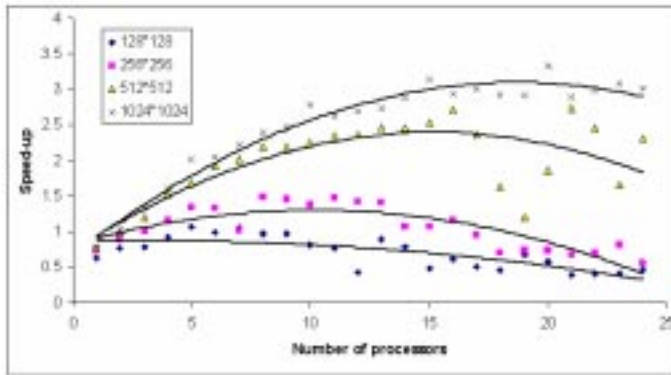


Fig. 8. Speed-up of the parallel watershed algorithm using PVM.

5 Conclusions

Brain/liquor ratio can be determined in a very robust manner with a watershed segmentation and a basin-joining method based on peaks in the histogram. The segmentation results are very accurate, as is shown in Figure 6. The speed-up and efficiency of the parallel watershed algorithm are not very good when PVM is used due to the many collisions on Ethernet when the resulting images are gathered. The speed-up for a CT slice is at most 3 on a 20 nodes machine while the efficiency is 20%. MPI is implemented on top of fast Myrinet and the results are better but still disappointing: a speed-up of at most 6 on 24 nodes with an efficiency of 30%.

This may lead to the conclusion that distributed memory systems are not a good choice for low level image processing and that the gathering of the labeled output image must be organized differently. A solution could be to use a routed network like Myrinet instead of Ethernet or to use SIMD processing. We can also use another distribution scheme of the image, for instance partition the image

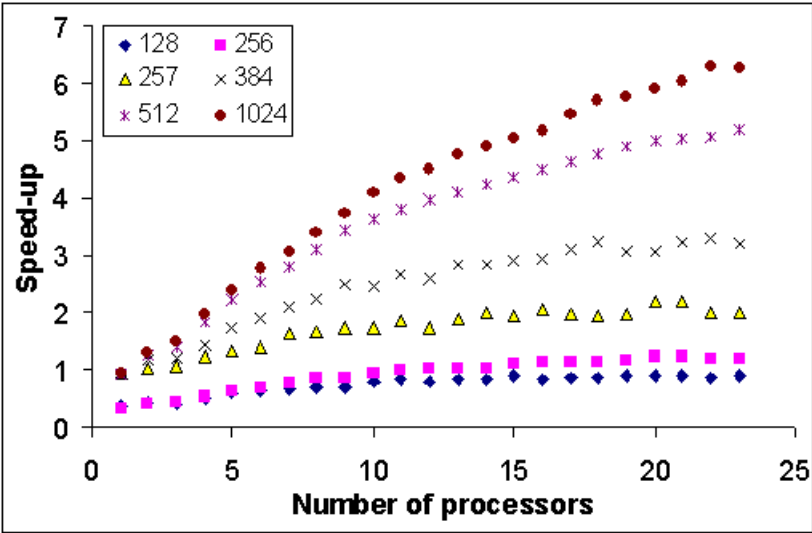


Fig. 9. Speed-up of the parallel watershed algorithm using MPI.

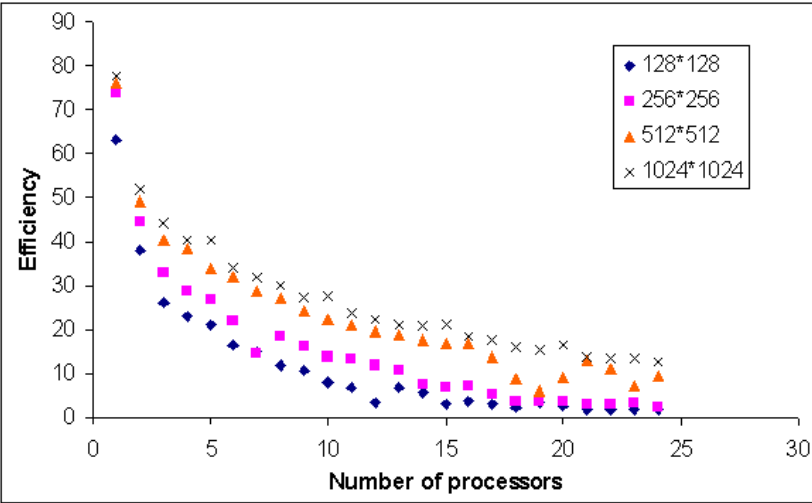


Fig. 10. Efficiency of the parallel watershed algorithm using PVM.

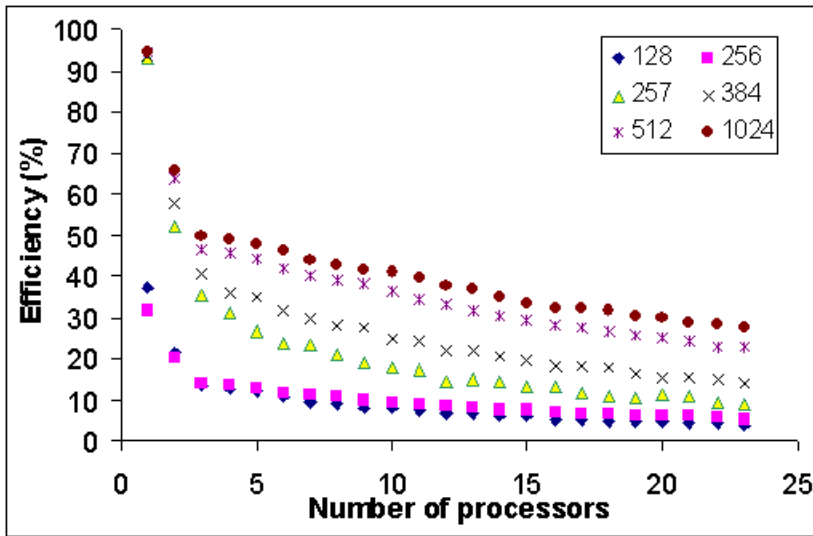


Fig. 11. Efficiency of the parallel watershed algorithm using MPI.

in non-equal parts, distribute each part to a different processor which will finish the processing at a different time and, in this way, minimize the collisions.

Acknowledgements

This work was sponsored by the EU as an activity of the TTN-Thuringen under contract EP 23713 TTN-T.

References

1. Vincent, L., Soille, P.: Watersheds in digital spaces: An efficient algorithm based on immersion simulations. *IEEE Trans. Patt. Anal. Mach. Intell.* **13**-(6) (1991) 583–598
2. Sera, J., Soille, P.: *Mathematical morphology and its applications to image processing*. Kluwer academic publishers (1994) 281–288
3. Moga, A., Cramariuc B., Gabbouj M.: An efficient watershed segmentation algorithm suitable for parallel implementation. *Proc. ICIP* **2** (1995) 101–104
4. Moga, A., Gabbouj, M.: A parallel marker based watershed transformation. *Proc. ICIP*. **2** (1996) 137–140
5. Meijer, E.J.: *Parallel region growing algorithms*. M.Sc. Thesis (1992) TU Delft
6. Meijster, A., Roerdink, J.: A proposal for the implementation of a parallel watershed algorithm. *Proc. CAIP* (1995) 790–795
7. Web page of HPISIS project <http://www.tnt.de>

This article was processed using the L^AT_EX macro package with LLNCS style

MPIPOV: A Parallel Implementation of POV-Ray Based on MPI*

Alessandro Fava¹, Emanuele Fava¹, and Massimo Bertozzi²

¹ Dipartimento di Ingegneria Industriale, Università di Parma
Parco Area delle Scienze 181/A, I-43100 Parma, Italy
fava@ce.unipr.it

² Dipartimento di Ingegneria dell'Informazione, Università di Parma
Parco Area delle Scienze 181/A, I-43100 Parma, Italy
bertozzi@ce.unipr.it

Abstract. The work presents an MPI parallel implementation of Pov-Ray, a powerful public domain ray tracing engine. The major problem in ray tracing is the large amount of CPU time needed for the elaboration of the image. With this parallel version it is possible to reduce the computation time or to render, with the same elaboration time, more complex or detailed images. The program was tested successfully on ParMa2, a low-cost cluster of personal computers running Linux operating system. The results are compared with those obtained with a commercial multiprocessor machine, a Silicon Graphics Onyx2 parallel processing system based on an Origin CC-NUMA architecture.

1 Introduction

The purpose of this work is the implementation of a distributed version of the original code of Pov-Ray [1], that is a well known public domain program for ray tracing. The parallelization of this algorithm involves many problems that are typical of the parallel computation. The ray tracing process is very complex and requires a notable quantity of computations that take, for the most complex images, from many hours up to days of processing. From that the need to increase the elaboration speed of these operations trying to compute the image in parallel.

We have used MPICH, a freely available, portable implementation of the MPI standard message-passing libraries [7], that allow to develop parallel programs on distributed systems, using standard programming language (C and Fortran) [3].

2 Ray Tracing with POV-Ray

POV-Ray (Persistence Of Vision Raytracer - www.povray.org) is a three-dimensional rendering engine. The program derives information from an external text file, simulating

* The work described in this paper has been carried out under the financial support of the Italian *Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST)* in the framework of the MOSAICO (Design Methodologies and Tools of High Performance Systems for Distributed Applications) Project.

the way the light interacts with the objects in the scene to obtain a three-dimensional realistic image.

The elaboration consists in projecting a ray for each pixel of the image and following it till it strikes an object in the scene that might determine the color intensity of that pixel. Starting from a text file that contains the description of the scene (objects, lights, point of view), the program can render the desired image. The algorithm works line by line with an horizontal scan of each pixel.

An interesting option of POV-Ray is the antialiasing. The antialiasing is a technique that helps to remove sampling errors producing a better image. Generally the antialiasing renders smoother and sharper images. Using the antialiasing option, POV-Ray starts tracing a ray for each pixel; if the color of the pixel differs from that of its neighborhood (the pixel on the left hand and that above), of a quantity greater than a threshold value, then the pixel is supersampled tracing a fixed number of additional rays. This technique is called supersampling and could improve the final image quality but it also increase considerably the rendering time. After the input data parsing, POV-Ray elaborates all the pixels of the image to render, trough an horizontal scan of each line from left to right. Once finished the elaboration of a line, this is written on a file or displayed on screen and then the following line is considered up to the last one (see Fig. 1).

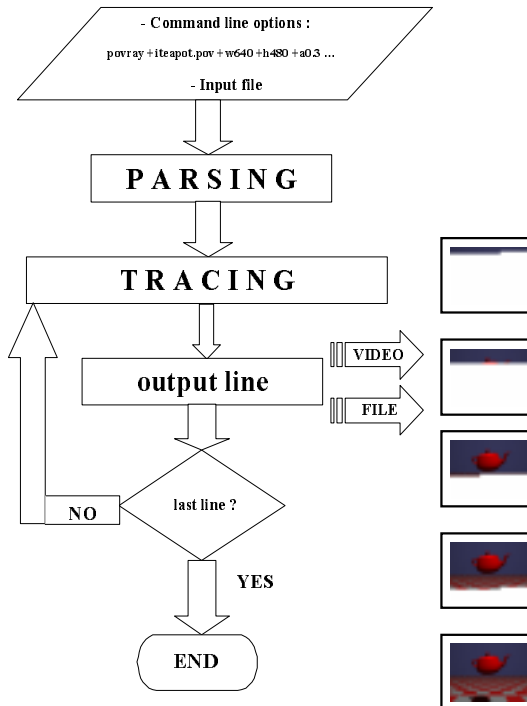


Fig. 1. Scheme of serial POV-Ray.

3 Parallel Implementation

Due to the high computational power required by POV-Ray, a parallel implementation is straightforward and other research groups tried to develop a parallel version of this application. The most famous is PVMPOV based on PVM [2, 6]. There are other porting based on MPI [4, 5] that, although deeply investigates the partitioning of the problem, seem not to face the complex problem of antialiasing feature of POV-Ray.

Conversely, the main goal of this work was obtaining an high performance parallel version of POV-Ray with all its features available (i.e. antialiasing).

Two different approaches have been investigated, the first (MPIPOV1) features a static division of the work determined before the run of the computation. This approach is suitable for homogeneous distributed architectures. The second approach (MPIPOV2) provides load-balancing assigning dynamically more work to faster nodes, thus fitting heterogeneous architectures.

The results, as concern the speedup have been very effective in both cases, even if the release with dynamic assignment is the worst from the point of view of the absolute performance, because of the larger number of communications needed respect to the static approach. This second release is potentially more efficient in the case of an heterogeneous cluster.

3.1 MPIPOV1: Static Parallelization

The first type of parallelization is based on a static partitioning of the work. Each node is assigned a section of the line to elaborate. The nodes elaborate their section in parallel and, once finished, they send it to a master process that, after processing its section, reconstructs the complete line and then display or save it on a file (see Fig. 2). This procedure is repeated up to the last line of the image.

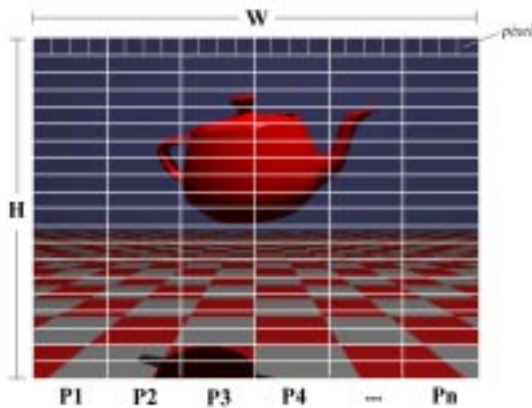


Fig. 2. Image subdivision in MPIPOV1: W = horizontal resolution (pixels), H = vertical resolution (pixels), n = number of processes, and $P1, \dots, Pn$ = processes.

Each line is divided in n sections of W/n pixels. There is a control for the possible remaining pixels forcing the last section to have the value W as upper limit. The complete image will result composed of n vertical strips of H sections of line. The master process receives $(n - 1) \times H$ sections while each slave sends H sections.

This implementation works fine on homogeneous parallel systems where each processing node features the same computation power. In fact a line is completed (and then written on disk or displayed on screen to continue the elaboration of the following) only when the slower process finishes the computation of its section while the faster processes remain idle, wasting useful time of elaboration. Moreover there is a small difference with the result produced by the original version: the antialiasing is, in fact, an intrinsically serial operation and so this function does not take into account the first pixels of each section that compose the complete line. This difference does not affect the image quality in noticeable manner.

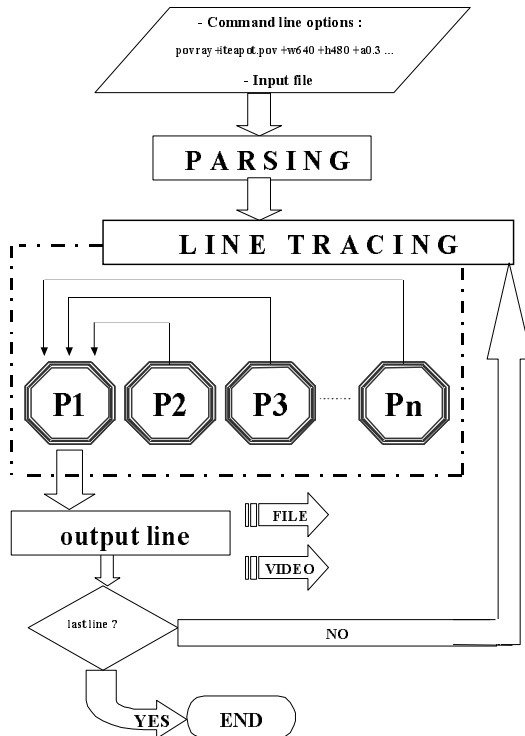


Fig. 3. MPIPOV1 Scheme.

3.2 MPIPOV2: Dynamic Parallelization

In the second approach the load of each process is dynamically balanced: in this way we assign more work to the faster processes: each process will elaborate a number of sections in proportion to its elaboration speed. In this way we try to find a solution to the slower process limitation, optimizing the use of each processor.

Each line of the image is now divided into a number of sections greater than the number of the available processes. Initially each processor is assigned a single section of the line to be computed; the master process, that does not have any section to elaborate, has the task to assign a new section to the first process that end its work. The elaboration goes on till no line sections are left. The procedure is iterated, also in this case, line after line (see Fig. 5).

The elaboration speed of each process could be different due to the internal characteristics of the architecture or to the machine load, but it also depends on the particular section of the image to process that could be more or less complex.

Now it is the user that can choose the number of vertical strips in which the image will be divided. The optimal value results difficult to find because it depends on many factors including the number of the available processes, the horizontal resolution and the machine load. In fact the lower grain of the computation can potentially improve the load balancing effect but also increase the communications.

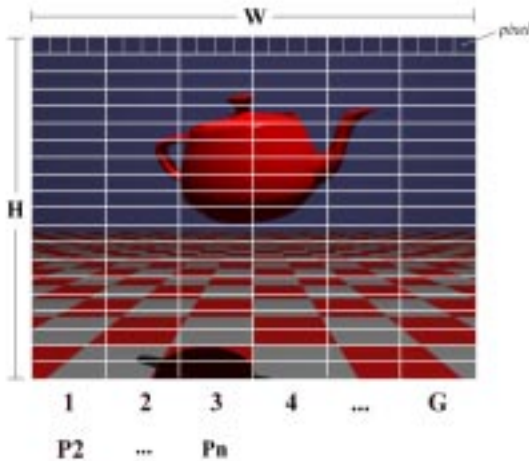


Fig. 4. Image subdivision in MPIPOV2: W = horizontal resolution (pixels) H = vertical resolution (pixels) n = number of processes $P1, \dots, Pn$ = processes G = number of section for each line.

Figure 4 shows more in detail how the image is now divided. The algorithm requires that the number of line sections G must be a value between $n1$ and W .

Each line is divided in: G sections of W/G pixels. There is a control for the possible remaining pixels forcing the last section to have the value W as upper limit.

The complete image will result composed from: G vertical strips of H sections of line.

The advantage obtained from the dynamic management of the sections to elaborate, that is appreciated only in cases of inhomogeneous load of the machines used, produces an expressive increase of the number of communications among the processes. The consequence is the increase of the time of broadcast and of synchronization due to the sending of the orders to the slaves and to the larger number of sections to manage. The number of communications increases further if the antialiasing option is used. In this case, each slave have to know the complete line previously elaborated and some information concerning the antialiasing of the same line; all this is sent from the master that reconstructs the various partial data received from the slaves during the computation of the individual lines. Also with this approach, remains the problem concerning the antialiasing of the first pixel of each line section.

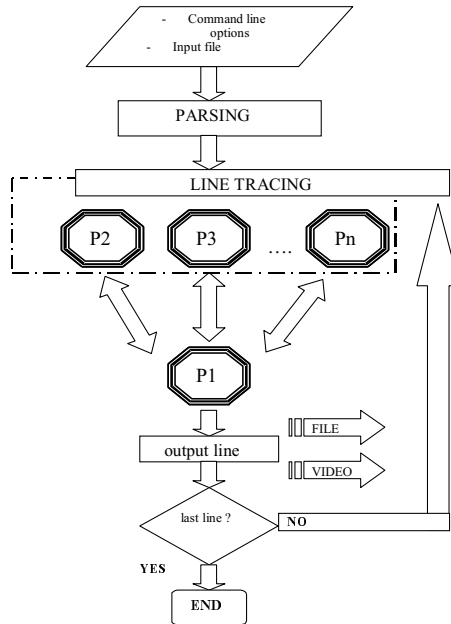


Fig. 5. MPIPOV2 Scheme.

4 Results

The examples has been tested on the distributed architecture ParMa², developed by Computer Engineering Department of the University of Parma [9]. ParMa² is a cluster of PC running Linux as operating system, composed of four Dual Pentium II 450 MHz interconnected by a 100 Mbit/s switched Fast Ethernet network, forming an 8-processor system.

The program is also been ported and tested on a commercial multiprocessor machine an SGI Onix2 (with 8 R10000 195 MHz processors).

The results obtained rendering the same sample image with different numbers of processors are presented in Tables 1 and 2.

# Processors	Time (sec.)		Speedup	
	ParMa ²	Onix2	ParMa ²	Onix2
1	60	76	1	1
2	30	40	2	1.9
3	24	31	2.5	2.45
4	19	23	3.15	3.3
5	15	20	4	3.8
6	12	16	5	4.75
7	10	14	6	5.42
8	9	12	6.66	6.33

Table 1. Speedup of MPIPOV1 on the Pc-cluster ParMa²and Onix2.

For MPIPOV1 we have for both architectures fairly linear speedup and for what concerns absolute performances the best time is obtained with the cluster ParMa².

# Processors	Time (sec.)		Speedup	
	ParMa ²	Onix2	ParMa ²	Onix2
1	60	76	1	1
2	67	80	0.89	0.95
3	37	41	1.62	1.85
4	28	29	2.14	2.62
5	24	22	2.5	3.45
6	23	18	2.6	4.22
7	21	16	2.85	4.75
8	20	15	3	5.06

Table 2. Speedup of MPIPOV2 on the Pc-cluster ParMa²and Onix2.

For MPIPOV2, that we can define communication-intensive, speedup and absolute performances are better for the Onix2 that can take advantage of the fast interconnection structure.

5 Conclusions

We have developed an efficient parallel version of POV-Ray, a well known and freely available ray tracing engine, which is particularly suited for the parallelization. The ray

tracing process is very complex and requires a notable quantity of computations and of time to obtain realistic three-dimensional images and so the need to increase the elaboration speed.

For the development of the parallel version, it has been used the MPI message-passing library, which can exploit the potential of low-cost architectures such as clusters of personal computers. The MPI parallel code is also easily portable on different architecture and operating system.

Two method has been described for parallel ray tracing. The first is a static algorithm that distributes equal work to each node of the parallel system, whereas in the second, the dynamic one, each node elaborates an amount of work in proportion to its elaboration speed.

The parallel release has been tested on a Linux based cluster (ParMa²) producing excellent results. Test included compares the use of static vs dynamic version and ParMa² cluster vs SGI Onix2 commercial multiprocessor machine. Final benchmark testing, using a well known sample image, yielded the best results using ParMa² system, showing the potentiality of the cluster architecture and therefore their efficiency in terms of cost/performance ratio with respect to traditional multiprocessor architectures.

A highly optimized dynamic version of POV-ray 3.1 that can also completely perform the antialiasing computation, is now under development [8].

References

1. O. Aftreth, G. Emery, and W. Morgan. The Online POV-Ray Tutorial, 1996. Available at <http://library.advanced.org/3285/>.
2. A. Dilger. PVM patch for POV-Ray, 1998. Available at <http://www-mddsp.enel-ucalgary.ca/People/adilger/povray/pvmpov.html>.
3. E. Fava. Realizzazione in Ambiente Distribuito di un Programma per Raytracing. Master's thesis, Università degli Studi di Parma - Facoltà di Ingegneria, 1999.
4. B. Freisleben, D. Hartmann, and T. Kielmann. Parallel Incremental Raytracing of Animations on a Network of Workstations. In *Procs. Intl. Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, volume 3, pages 1305–1312, July 1998.
5. B. Freisleben, D. Hartmann, and T. Kielmann. Parallel Raytracing: A Case Study on Partitioning and Scheduling on Workstation. In *Procs. 30th Hawaii Intl. Conf. on System Sciences*, volume 1, pages 596–605, Jan. 1998.
6. A. Haveland-Robinson. POVbench home page, Apr. 1999. Available at <http://www.haveland.com/povbench/>.
7. Mathematics and Computer Science Division, University of Chicago, Chicago. *User's guide for MPICH, a Portable Implementation of MPI*, 1995.
8. R. Sbravati. Parallellizzazione di POV-ray 3.1 utilizzando la libreria MPI. Technical report, Dipartimento di Ingegneria dell'Informazione, Università di Parma, June 1999.
9. D. Vignali. ParMa² White Paper. Technical report, Dipartimento di Ingegneria dell'Informazione - University of Parma, Sept. 1998. Available at <ftp://ftp.ce.unipr.it/pub/ParMa2>.

Minimum Communication Cost Fractal Image Compression on PVM

Pou-Yah Wu

Department of Mathematics Education
National Tainan Teachers College, Tainan 700, Taiwan
pywu@ipx.ntnct.edu.tw

Abstract. In this paper, we propose a method of regional search for fractal image compression and decompression in a PVM system. In this method, the search for the partitioned iterated function system (PIFS) is carried out in a region of the image instead of over the whole image. Because the area surrounding of a partitioned block in an image is similar to this block possibly, finding the fractal codes by regional search results in increased compression ratios and decreased compression times. When implemented on the PVM, the regional search method of fractal image compression has the minimum communication cost. We can compress a 1024 x 1024 Lenna's image on a PVM with 4 Pentium II-300 PCs in 13.6 seconds, with a compression ratio 6.34; by comparison, the conventional fractal image compression requires 176 seconds and has a compression ratio 6.30. In the future, we can apply this method to fractal image compression using neural networks.

1 Introduction

Today, computer applications such as distance diagnostic medical system, digital libraries and Internet are steadily gaining in popularity. One of the most important problems in such applications is how to store and transmit images. Image compression reduces both memory storage and the transmission time [11]. Previous developed compression techniques include the discrete cosine transform technique and pyramid coding. In this paper, we study a more recently developed method of compression, fractal image compression.

Fractal image compression is based on self-similarity property of an image, and performs image compression by applying a series of transformations. These transformations are applied to perform image decompression iteratively until the system converges. Previous workers have used Hutchinson metric, and Mandelbort one of the first worker to generate images using the fractal theory [1], [2], [10]. Barnsley developed the collage theorem and the iterated function system (IFS), and was able to produce a method for fractal coding, which greatly enhanced compression ratios [1], [2], [3], [4]. Jacquins improved on IFS by proposing the partitioned iterated function system (PIFS) which is able to automatically determine the fractal code [7],

[8]. The main disadvantage of previous techniques for fractal image compression is the large amount of computation that they require [5].

A number of more efficient methods have been proposed to achieve higher compression ratios and smaller compression times. Efforts to increase the compression ratio have been based on alternative methods of partitioning [12], alternative encoding methods [7], and heuristic search methods [13]; attempts to decrease compression time have also resorted to a variety of techniques, including image classification [5], parallel processing [14], and processing using neural networks [9].

Parallel processing, a method in which many small tasks are used to solve one large problem, has been facilitated by two major developments: massively parallel processors (MPPs) and the widespread use of distributed computing. Distributed computing offers many advantages; for instance, by using existing hardware, the cost of computing can be significantly reduced. In addition, the virtual computer resource that is created by distributed computing can grow in stages and can thus take advantage of the latest computational and network technologies. As distributed computing and MPP have gained in popularity the notion of message passing has also become increasingly common. The Parallel Virtual Machine (PVM) system uses a message-passing model to allow programmers to take advantage of distributed computing, using a wide variety of computer types, including MPPs. To be effective, distributed computing requires high communication speeds [6].

We have previously studied some parallel methods for fractal image compression using a hypercube [14]. In these studies, we parallelized sequential fractal image compression by domain decomposition on a message-passing iPSC/860 multicomputer with 8 nodes. We also parallelized a neural network algorithm for decoding IFS codes on a hypercube. One problem encountered in these studies was high communication costs.

In this study, we parallelize the fractal image compression using regional search on the PVM so that the communication costs are minimized. In Section 2, we present a technique for sequential fractal image compression using regional search. In Section 3, we present the results of studies using this technique and discuss its impact on communication costs; in Section 4, we present our conclusion.

2 Fractal Image Compression with Regional Search

In this section, we develop a new search strategy for fractal image compression. Searching for the fractal codes is one of the major tasks of fractal image compression, and there are several directions that we pursue as we seek to improve the compression process. First, we can design a search method which will reduce the execution time of image compression. Secondly, we can seek to reduce the loss ratio. Finally, we can aim to improve the compression ratio.

In an image, the area which surrounds a partitioned block is naturally similar to the block. Therefore, a search which is prioritized by nearness may be quicker than a search which begins from the upper left corner of the image. Prioritizing the search by

nearness also improves the compression ratio at the same time. Although the compression method used in this study classifies domains according to the archetype, it starts to search the fractal codes from the same archetype class that is permuted from the upper left corner to the lower right corner of the whole image.

However, assuming that we wish to compare the nearest domain with the range each time, we find that this requires a lengthy execution time. In the regional search method, the original image is partitioned into nonoverlapping regions with sufficiently large size as shown in Fig. 1. In this method of fractal image compression, the search takes place in similar domains from the same region of the range. Simultaneously, we also use the archetype classification in the regional search.



Figure 1. The regional search.

The region R partitioned by the original image G is a nonempty compact subset of G . Hence R is also a subset of a complete metric space (X, d) . Fisher [5] has shown that there exists a set of contractive affine transformations $W : H(X) \rightarrow H(X)$ such that the region R becomes a fixed point of W , i.e. $W(R) = R$. According to PIFS theory, the union of these sets of contractive affine transformations has the original image G as the fixed point. The algorithm for the fractal image compression using regional search is shown in Algorithm 1.

Similarly, the algorithm for fractal image decompression using regional search is shown in Algorithm 2.

Algorithm 1. Algorithm for the regional search method of fractal image compression.

Input : An original image G
 Output : The compressed file of the image
 Method :
 Choose a partition number n and a tolerance level ϵ_c ;
 Partition G into nonoverlapped regions G_0, G_1, \dots, G_{n-1} ;
 For each G_j do {

```

Set  $R_i = G_j$  and mark it uncovered;
While there are uncovered ranges  $R_i$  do {
Find the domain  $D_i$  and the corresponding  $w_i$  that best cover  $R_i$ ;
    If  $d(R_i, w(D_i)) < e_c$  or  $\text{size}(R_i) < r_{\min}$  then
        Mark  $R_i$  as covered, and write out the transformation  $w_i$ ;
    Else
        Partition  $R_i$  into smaller ranges that are marked as uncovered,
        and remove  $R_i$  from the list of uncovered range;
    }
}

```

Algorithm 2. Algorithm for fractal image decompression using regional search.

Input : The fractal image compression code

Output : The image decompression file

Method :

```

Choose any initial image  $G_0$  with the same size of the region of the original
image as the region of the domain image and an iteration number  $i_c$ ;
For each region do {
    Apply the fractal image code to the region of the domain image
    iteratively
    until the iteration number is  $i_c$ ;
}
Output the transformed image as the image decompression file;

```

3 Minimum Communication Cost Fractal Image Compression on PVM

Now, we parallelize the regional search of fractal image compression by using PVM. We assume that the ratio of the byte number $p \times p$ of the image and the node number n of PVM greater than 2^{10} . Since the issue of performance on PVM is primarily concerned with communication overhead, the main objective of our design is to reduce communication cost.

Fractal image compression partitions the original image into n regions and sends these regions to nodes. Each node performs conventional fractal image compression on a region. Then, the fractal codes of each region are sent to the master node and are merged to a compressed file. The algorithm is shown in Algorithm 3.

Similarly, the master program for the regional search method of fractal image decompression using sends the fractal codes for the different regions to child nodes. Then, the master and the slave node apply the fractal codes for a given region to a region of the same size in any initial image iteratively. After decompression of all the regions are complete, the slave program sends the decompressed regions to the master and they are merged to a decompressed file. The algorithm is shown in Algorithm 4.

Our algorithm for the master program contains only two steps which require communication on the PVM. Before finding the PIFS for the regions, we must send image data for the regions to child nodes. After the fractal codes for the regions are

found, we must send them to the master. Similarly, the communication costs involved in the algorithm for the slave program reflect a minimum level of communication within the PVM. Hence, our algorithm minimizes communication costs, as shown in Theorem 5.

Algorithm 3. Algorithm for the master program of fractal image compression.

Input : An original image G and a node number n

Output : The compressed file of the image G

Method :

```

    Choose a tolerance level  $e_c$ ;
    Partition  $G$  into nonoverlapped regions  $G_0, G_1, \dots, G_{n-1}$ ;
    Broadcast the slave program to each child node;
    Broadcast the tolerance level and the size of the region to each child node;
    For each child  $j$  do { send  $G_{j+1}$  to the child node  $j$ ; }
    Set  $R_1 = G_0$  and mark it uncovered;
    While there are uncovered ranges  $R_i$  do {
        Find the domain  $D_i$  and the corresponding  $w_i$  that best cover  $R_i$ ;
        If  $d(R_i, w(D_i)) < e_c$  or  $\text{size}(R_i) < r_{\min}$  then
            Mark  $R_i$  as covered, and record the transformation  $w_i$ ;
        Else
            Partition  $R_i$  into smaller ranges that are marked as uncovered, and
            remove  $R_i$  from the list of uncovered range;
    }
    For each child  $j$  do { Receive the transformations from the child node  $j$ ; }
    Write out the transformations;
    Exit PVM;

```

Algorithm 4. Algorithm for slave program at node j of fractal image compression.

Method :

```

    Receive the tolerance level  $e_c$  and the size of the region from the master;
    Receive  $G_{j+1}$  from the master;
    Set  $R_1 = G_{j+1}$  and mark it uncovered;
    While there are uncovered ranges  $R_i$  do {
        Find the domain  $D_i$  and the corresponding  $w_i$  that best cover  $R_i$ ;
        If  $d(R_i, w(D_i)) < e_c$  or  $\text{size}(R_i) < r_{\min}$  then
            Mark  $R_i$  as covered, and record the transformation  $w_i$ ;
        Else
            Partition  $R_i$  into smaller ranges that are marked as uncovered,
            and remove  $R_i$  from the list of uncovered range;
    }
    Send the recorded transformations to the master;
    Exit PVM;

```

Theorem 5. The communication cost of the fractal image compression using regional search is minimum.

Proof. Assume that the size of processed image be t and the size of region processed by the master program be m . Because the master program must send and receive the image data with the size $t-m$ to and from child nodes, the lower bound of communication cost of distributed fractal image compression is $2(t-m)$. It is clear, the communication cost of the fractal image compression using regional search is the lower bound.

Algorithm 6. Algorithm for the master program of fractal image decompression.

Input : The fractal image compression code and an iteration number i_c

Output : The image decompression file

Method :

Broadcast the slave program to each child node;

Broadcast the region size and the iteration number to each child node;

For each region do

{Send the fractal codes on the region to the child node;}

Choose any initial image G_0 in which the regions are the same size as the regions of the domain image ;

Apply the fractal image code for a region of the initial image to the corresponding region of the domain image iteratively until the iteration number is i_c ;

Record the transformed region of the image;

Receive the transformed region of the image from each child node;

Output the transformed image as the image decompression file;

Exit PVM;

Algorithm 7. Algorithm for the slaver program of fractal image decompression.

Method :

Receive the region size and the iteration number from the master;

Receive the fractal codes for the respective region from the master;

Choose any initial image G_0 with the regions having the same size as the respective regions of the domain image ;

Apply the fractal image code for a region of the original image to the corresponding region of the domain image iteratively until the iteration number is i_c ;

Record the transformed region of the image;

Send the transformed region of the image to the master;

Exit PVM;

Our PVM system is composed of 4 Pentium II-300 PCs connected by 100Mbps/sec Ethernet. We ran the fractal image compression by the Lenna's image with a 1024×1024 image size as shown in Fig. 2(a).

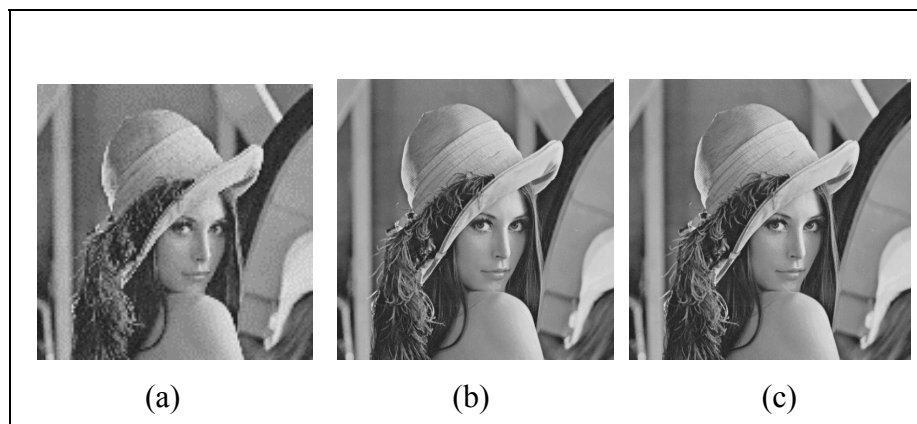


Figure 2. (a) The original image.

(b) The decompressed image by the conventional fractal image compression.

(c) The decompressed image by the fractal image compression using local search on PVM.

First, we compressed the Lenna's image using the conventional fractal image compression. Compression time was 176 seconds and the compression ratio was 6.30. The decompressed image from the conventional fractal image compression is shown in Fig. 2(b) [11].

Next, we compressed the same image using regional search on PVM with 4 nodes. Compression time was 13.6 seconds and the compression ratio was 6.34. The decompressed image from this method is shown in Fig. 2(c).

4 Conclusions

This paper proposes a method for fractal image compression at minimum communication cost on the PVM. Question to be addressed in the future is how to be implemented by the hardware to reduce communication cost. In addition, we plan to research methods of reducing the loss ratio and increasing the compression ratio in parallel fractal image compression.

References

- [1] M. F. Barnsley, *Fractal everywhere*, New York, Academic Press , 1992.
- [2] M. F. Barnsley and A. D. Sloan, *A better way to compress images*, Byte, vol.13 (1988) 215-223.
- [3] M. F. Barnsley, A. Jacquin, L. Renter and A. D. Sloan, *Harnessing chaos for image synthesis*, Computer Graphics, vol.22, no.4 (1988) 131-141.
- [4] S. Demko, L. Hodges and B. Nayloy, *Construction of fractal objects with iterated function systems*, Computer Graphics, vol.19, no.3 (1985) 271-278.
- [5] Y. Fisher, *Fractal image compression: Theorem and Application*, New York, Spring-Verlag (1994)
- [6] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, *PVM: Parallel Virtual Machine-A Users' Guide and Tutorial for Networked Parallel Computing*, The MIT Press, Cambridge, Massachusetts London, England, 1994.
- [7] A. E. Jacquin, *Image coding based on a fractal theory of iterated constructive image transformations*, IEEE Trans. on Image Processing, vol.1, no.1 (1992) 18-30.
- [8] A. E. Jacquin, *Fractal image coding: a review*, Proceeding of the IEEE, vol.81, no.10 (1993) 1451-1465.
- [9] S. L. Lee, P. Y. Wu and K. T. Sun, *Fractal Image Compression Using Neural Networks*, IEEE International Joint Conference on Neural Networks, Anchorage, Alaska, May 5-9 (1998) 613-618.
- [10] B. Mandelbort, *The fractal geometry of nature*, San Francisco CA: freeman, 1982.
- [11] M. Nelson, *The data compression book*. 2nded., M&T books, New York, 1996.
- [12] E. Shusterman and M. Feder, *Image Compression via improved quadtree decomposition algorithm*, IEEE Trans. on Image Processing, vol.4, no.6 (1994) 207-215.
- [13] L. Thomas and F. Derrai, *Region-based fractal image compression using heuristic search*, IEEE Trans. on Image Processing, vol.4, no.6 (1995)832-838.
- [14] P. Y. Wu, *A Study of The Parallelization of Fractal Image Compression Using 8×8 Partition on a Hypercube*, Journal of National Tainan Teachers College, vol.31 (1998) 283-296.

Cluster Computing Using MPI and Windows NT to Solve the Processing of Remotely Sensed Imagery

J. A. Gallud¹, J.M. García², and J. García-Consuegra¹

¹ Departamento de Informática,
Universidad de Castilla-La Mancha,
Campus Universitario, 02071 Albacete, Spain
{jgallud, jdgarcia}@info-ab.uclm.es.

² Departamento de Ingeniería y Tecnología de Computadores,
Facultad de Informática. Universidad de Murcia, Spain,
Campus de Espinardo, 30080 Murcia
jmgarcia@dittec.um.es

Abstract. The design of efficient distributed applications depends on the coordinate use of different API (Application Programming Interface) like MPI and NT API's. In fact, a particular optimized code can be reused in many other applications reducing the cost of its design by means of a set of libraries. Distributed processing is applied in remote sensing in order to reduce spatial or temporal cost using the message passing paradigm. In this paper, we present a workbench called DIPORSI, developed to provide a framework for the distributed processing of Landsat images using a cluster of NT workstations. Our application is based on a NT implementation (WMPI) of the MPI standard. Thus, the large amount of time required by the sequential processes drops when the parallel processing is used. Moreover, we have obtained a reduction of computation time over the 400% for large size images and a moderate number of parallel nodes. Our results confirm that cluster computing is a cost/performance effective solution to the remotely sensed image processing.

1 Introduction

The available of high-speed networks and increasingly powerful commodity processors is making the usage of clusters of computers an appealing vehicle for cost effective parallel computing. Cluster computers have several advantages, as they are built using commodity-off-the-shelf hardware components and they can be programmed using the standard parallel tools and utilities. Clearly, the cluster environment is better suited to applications that are not communication-intensive, since a LAN typically has a high message start-up latencies and low bandwidths.

However, several research projects are analyzing the communication subsystem, in order to provide the same quality of message delivery as MPPs. The goal

of these projects is to reduce the message latency to the minimum, usually by eliminating the most of the operating system and device driver overheads [3, 14]

Traditionally, parallel processing is an area that it has been dominated by Unix-based systems. That is also true in the cluster computing field. However, the computers PC-based market is clearly dominated by the Windows NT operating system. Recently, many research projects have explored the possibilities of cluster computing under Windows NT [2, 4].

Remote sensing involves the manipulation and interpretation of digital images which have been captured from remote sensors on board of satellite or aircraft systems. Such images collect information about the Earth's surface, which allow scientists to perform many environmental studies.

Remotely sensed image processing is an interesting application area for distributed computing techniques [8, 13]. The large data volumes involved and the consequent processing bottleneck may indeed reduce their effective use in many real situations, and hence the need for exploring the possibility of splitting both the data and processing over several storing and computing units [1]. All the procedures involved in the digital image manipulation of such images may be categorized into one or more of the following four broad types of computer assisted operations: image rectification and restoration, image enhancement, image classification and data merging [11].

In this paper, we describe a distributed workbench called DIPORSI, by means the coordinated use of both the MPI API and the Windows NT File System API. DIPORSI stands for *DIstributed Processing Of Remotely Sensed Imagery* and has been designed to run on a cluster of Windows NT workstations. DIPORSI was designed to perform a considerable number of the former tasks by using a cluster of workstations composed by NT platforms which are connected by means of an Ethernet network using the Message-Passing Interface standard (MPI). MPI provides an interface to design distributed applications that run on a parallel system [7].

This paper shows how reducing the long computation time required by remote sensing distributed procedures. All the distributed applications are implemented by splitting the original images into several small ones, which are processed in each node simultaneously. Moreover, remote sensed image is a good application to run in a distributed way, because it is computation-intensive with a small communication between processes. The comparative results have been obtained using a distributed algorithm to georeference a distorted remotely sensed image. This algorithm has been coded in MPI, and we have obtained a very good speedup, and a reduction in the computation time over the 400% for large images and a moderate number of nodes.

The following section explains the structure of the current DIPORSI workbench. In section 3 we present the comparative study based on a particular processing of Landsat-TM images. Finally, in section 4 the conclusions and future work are drawn.

2 The DIPORSI Structure and Its Capabilities

DIPORSI workbench was designed to perform in a distributed way, a great variety of the algorithms used in the remotely sensed image processing. These algorithms work to obtain either a new digital image or new features from the original image.

Many remote sensing algorithms work in a parametric way, that is, the requested application is related to the initial parameters. The newer and most used algorithms are the classifiers and their behaviour are governed by a set of initial parameters. To obtain a classified image, the algorithm begins its computations with such initial values as yield the resultant image. This resultant image depends on both the initial parameters as well as the original multidimensional image. In many cases, the process must be repeated with other parameter values because the results are unacceptable. This explains both the need of reducing the spatial and temporal costs by means of a distributed processing, and the parametric nature of the distributed workbench.

DIPORSI appears as a layer between MPI functions and the code of each process. DIPORSI offers a set of functions and a message structure to the user for performing in an easy and distributed manner whatever remote sensing algorithm. Some recent works have proposed similar frameworks for related problems [5, 9].

In figure 1, the functional diagram of DIPORSI can be seen. Note that only one process is executed in each processor.

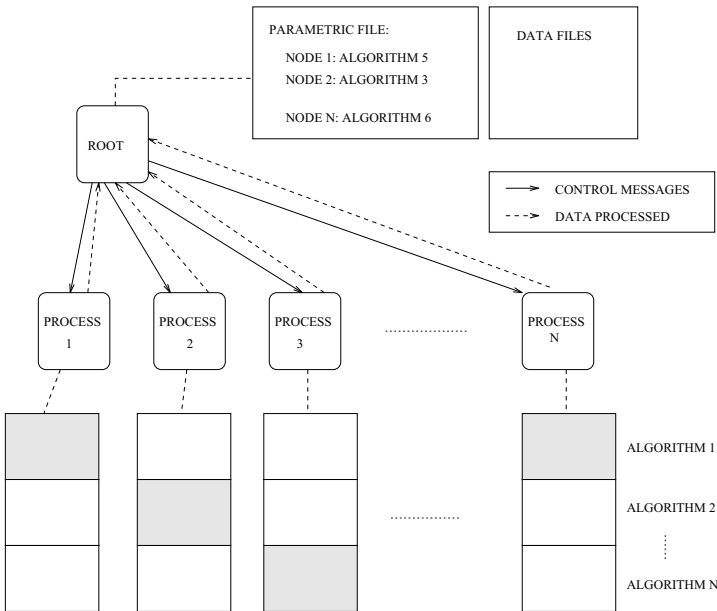


Fig. 1. DIPORSI schema

DIPORSI runs in a batch way. Thus, all the user has to do is to generate a parametric file with the following information: The algorithm (or the sequence of algorithms) to manipulate the remote sensed image (i.e. the georeferencing algorithm using the bilinear interpolation method), the workload distribution to the computing nodes, and the number and the location of the data images.

DIPORSI provides the user a number of functions for managing files of different kinds such as text files, data files, raster images, etc. Our application allows us to work by distributing both spatially and temporarily either of the remote sensing algorithms. That is, we can make all the nodes perform the same computations on different data or each node does different computations on the same image. Also, DIPORSI generates special messages to the user in response to execution of the distributed algorithm.

Initially, DIPORSI was implemented with a set of functions to manage files to allow that processes manage the data transfer control. These functions were designed by means of MPI functions to send and receive data and control files like images or the parametric files. The idea was based on providing flexibility to individual processes. However the response times were affected by the task of broadcasting the images among the nodes. Currently, DIPORSI uses the file system functions provided by NT operating system instead of calling a function implemented from MPI primitives. Thus, the distributed processes can make use of the network capabilities present in the NT operating system like protection and security.

3 The Performance of DIPORSI

In this section we present the comparative results between DIPORSI versus the sequential execution of a particular algorithm. For comparison purposes, we run a remote sensing algorithm that is used to compute a corrected image from a distorted one. Next, the algorithm used will be briefly described. In [6] we presented a preliminary results we have obtained with the initial implementation of DIPORSI.

When a image is remotely sensed, a number of different errors appear distorting it in such a way that cannot be used correctly. The process of georeferencing a satellite image consists in the application of a set of mathematical operations on the original image to obtain a geometrically corrected image. So, the purpose of geometric correction is to compensate for the distortions introduced by different factors (earth curvature, relief displacement, and so on) so that the corrected image will have the geometric integrity of a map [11, 12].

The usual procedure applies the traditional polynomial correction algorithm with or without using a digital model terrain. This kind of distortion is corrected by analyzing well-distributed ground control points occurring in an image. These values are then submitted to a least-squares regression analysis to determine coefficients for two coordinate transformation equations that can be used to interrelate the geometrically correct coordinates and the distorted image coordinates. Once the coefficients for these equations are determined, the

distorted image coordinates for any map position can be precisely estimated. However, the process is actually made inversely. An undistorted output matrix of empty map cells is defined, and then each cell is filled in with the gray level of the corresponding pixel (digital number or DN), or pixels depending on the method employed, in the distorted image [10]. That is to say, the process is performed using the following operations:

1. The coordinates of each element in the undistorted image are transformed to determine their corresponding location in the original distorted image.
2. The intensivity value or DN of the undistorted image is determined by using one of these usual methods: nearest neighbour, bilinear interpolation and cubic convolution.

The distributed algorithm works in a similar way as the sequential. The first step is made by the master process, which opens a file that contains the information of the task to be solved. This process reads a parametric file, in which all the activities to be performed in the distributed environment and their parameters are specified in an ordered way. The master process sends such information to all the nodes involved. Thus, each node knows what it must do (correction method) and how much information it must receive (number of bands -the resultant image to a given frequency-, resolution, the transformation functions, etc).

The next step consist of each node runs the algorithm, which acts on the region of the distorted image where the computations must do the corrections. Each node uses the file system functions to access to the data and compute its partial sub image locally.

The last step consist on the task of recovering the resultant image. In the older algorithm this was made by the root process. In this distributed algorithm, each node writes its partial sub image directly on the main node by means of file system functions.

A Landsat image has 7 bands, 6 with 30x30 meters and 1 with 120x120 meters of resolution respectively, with approximately 40Mb each band, which explains the high value of the response time when geometric correction is computed in a single machine. Our parallel algorithm works by splitting the original distorted image into number-of-rows/number-of-nodes blocks in accordance with an uniform workload allocation, then each node computes a small submatrix reducing the computation time and improving the overall performance of the algorithm.

Our hardware environment is the following. The single machine is a Pentium II 333 Mhz with 32 MB of RAM running Windows NT Workstation 4.0. The distributed machine is composed of the 8 PII 333 Mhz with 32 MB of RAM running Windows NT Workstation v4.0. The nodes are linked using a 10 Mbps Ethernet.

The figure 2 shows the comparative results obtained between the sequential and the distributed algorithm. The times of the distributed algorithm were obtained with two nodes. It can be observed that the larger image sizes are used, the better results are achieved with the distributed algorithm.

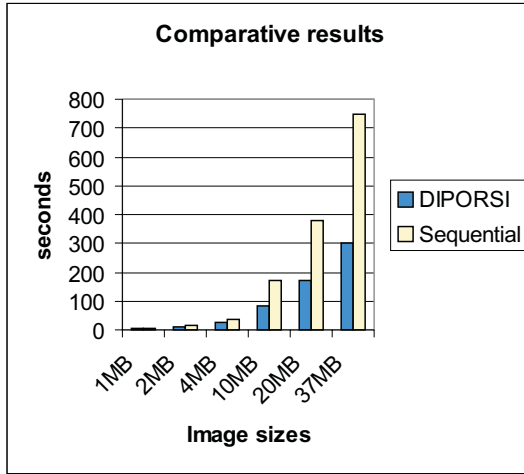


Fig. 2. The distributed algorithm vs sequential.

In the figure 3 we can see the speedup obtained with the distributed algorithm for a range from 2 to 8 nodes. Obviously, not all the nodes show the exactly same behaviour though are quite similar.

The figure shows the linear reduction of the time as many as the number of the nodes increase. Moreover, it can be seen that we achieve a near linear speedup for large image sizes.

However, these figures show the computation times, not the total response time of the algorithm. This value is worse due to the communications overhead by transferring the images through a slow interconnection network. Amdahl's law implies that the speedup obtained from faster processors is limited by slowest system component; so, it is necessary to improve the network performance such that it balances with CPU performance. Therefore, we can evaluate DIPORSI with a fast interconnection network like fast Ethernet or Myrinet. Another possibility is to distribute image files in several hard disks. It would be supported by a parallel file system based on software RAID. In this way, we could improve the I/O performance by means of carrying out I/O operations also in parallel.

4 Conclusions and future work

This paper describes our distributed workbench called DIPORSI that has been designed to execute the main remote sensing algorithms using a cluster of workstations.

Our distributed algorithm has been coded with MPI. MPI is very useful for implementing distributed applications on low-cost platforms, specially suitable in the remote sensing area.

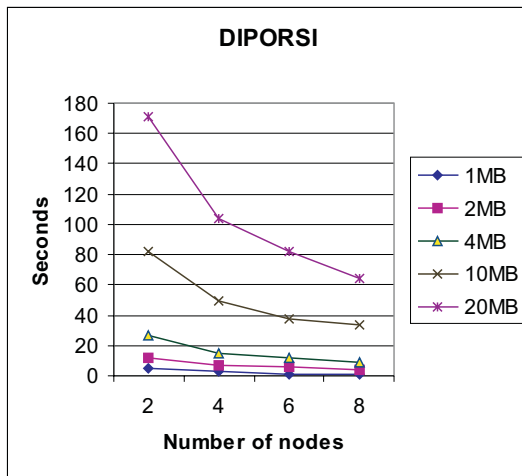


Fig. 3. Results obtained with the distributed algorithm (only computations).

The implementation shows the timing results when different image sizes are used. These results show a nearly linear speedup for large image sizes, and a reduction in the execution time over the 400% for a moderate number of nodes.

Future work admits of several possibilities: the evaluation of DIPORSI with fast networks and software RAIDs, the implementation of other algorithms to solve geometric correction, and finally, the integration with Unix platforms.

References

1. ANDERSEN, J.D., *Digital Image Processing: A 1996 Review*, Applied Parallel Computing - 3rd International Workshop Para 96, LNCS 1184, Springer-Verlag (1996).
2. BAKER, M., *MPI on NT: The Current Status and Performance of Available Environments*, EuroPVM-MPI'98, LNCS No. 1497, Springer-Verlag, (1998), pp 63-75.
3. CULLER, D., LIU, L.T., MARTIN, R.P., YOSHIKAWA, C.O., *Assessing Fast Network Interfaces*, IEEE Micro, 16(1), (1996), pp 35-43.
4. DASGUPTA, P., *Parallel Processing with Windows NT Networks*, Proc. of the USENIX Windows NT Workshop, August (1997).
5. FOSTER, I., GEISLER, J., GROPP, W., KARONIS, N., LUSK, E., THIRUVATHUKAL, G., TUECKE, S., *Wide-area Implementation of the Message Passing Interface*, Parallel Computing, No. 24, (1998), pp 1735-1749.
6. GALLUD, J.A., GARCÍA-CONSUEGRA, J.D., SEBASTIÁN, G., *Distributed Georeferencing of Remotely Sensed LandSat-TM Imagery Using MPI*, Para'98, LNCS No. 1541, Springer-Verlag (1998), pp 161-167.
7. GROPP, W., LUSK, E., SKELLUM, A., *Using MPI Portable Parallel Programming with the Message Passing Interface*, The MIT Press, (1994).
8. HOFFMAN, F.M., HARGROVE, W.W., *Multivariate Geographic Clustering Using a Beowulf-style Computer*, PDPTA'99, Las Vegas (USA), (1999).

9. LEE, C., HAMDI, M., *Parallel Image Processing Applications on a Network of Workstations*, Parallel Computing, No. 21, (1998), pp 137-160.
10. LILLESAND, T.M., KIEFER, R.W., *Remote Sensing and Image Interpretation 2nd Edition*, J.Wiley & Sons.
11. MARKHAM, B.L., *The Landsat Sensors' Spatial Responses*, IEEE Transactions on Geoscience and Remote Sensing, Vol. GE-23 No. 6, (1986).
12. MATHER, P.M. *Computer Processing of Remotely-Sensed Images*, John Wiley & Sons.
13. McCORMICK, J.A., ALTER-GARTENBERG, R., HUCK, F.O., *Image Gathering and Restoration: Information and Visual Quality*, Journal of Optical Society of America, Vol 6 No. 7, (1989), pp 987-1005.
14. PIERNAS, J., FLORES, A., GARCÍA, J.M., *Analyzing the Performance of MPI in a Cluster of Workstation Based on Fast Ethernet*, EuroPVM-MPI'98, LNCS No. 1497, Springer-Verlag (1998), pp 63-75.

Ground Water Flow Modelling in PVM*

L. Hluchý, V. D. Tran, L. Halada, and M. Dobrucký

Institute of Informatics, Slovak Academy of Sciences
Dúbravská cesta 9, 842 37 Bratislava, Slovakia
upsyhluc@savba.sk

Abstract. In this paper we present the ground water flow modelling in PVM. It requires the solution of a partial differential equation representing a 3-D diffusion process. Its numerical solution by a finite difference method leads to a very large sparse, well-structured system of linear algebraic equations. The Strongly Implicit Procedure (SIP) algorithm is chosen to solve the matrix. We also present our parallel version of SIP on PC cluster with PVM. The experiments show that our parallel SIP is very effective.

Keywords: Parallelization, ground water flow model, finite difference method, strongly implicit procedure.

1. Introduction

Planning and management of ground water resources, environmental protection of ground water recharge including pollution control and measures, require a tool for predicting the response of the aquifer system and processes in the aquifer system to the planned activities.

The tool for achieving such comprehensive goals is the model, which can represent the real nature processes by a system of partial differential equations. However, the real natural aquifer system and the water quality processes in this system are very complicated. The best-suited modelling tools are the numerical methods where differential equations are replaced by a set of algebraic equations of the state variables at the discretization points in space and time [4], [5]. Thus, the problem is that the number of the algebraic equations to be solved simultaneously is very large and to reach the required accuracy of decision, rather large computations are required.

Solving diffusion and convection differential equations, which describe the fluid flow, is one of the most widely studied problems in scientific computations. With the arrival of parallel computers, this became possible for large 3D tasks as well. The effort to find stable and efficient parallel algorithms and/or program codes for various types of parallel computers became of primary interest for various research groups.

There are several numerical methods allowing to solve these equations. One is the finite difference method, which proved to be suitable for numerous 2D and 3D applications. Application of this technique results in solving a large system of linear

* This work was supported by the Slovak Scientific Grant Agency under Research Project No.2/4102/99.

equations $Ax=b$, approximately of the order 10^5 – 10^6 , where the system matrix is sparse, symmetric and quasi-bounded. This is not only very time consuming but also requires large physical memory to contain the matrix, so that parallel methods are of great importance.

2. The Mathematical Model and Its Discretization

The fluid flow model can be described by the partial differential equation:

$$\frac{\partial}{\partial x}\left(K_x \frac{\partial H}{\partial x}\right) + \frac{\partial}{\partial y}\left(K_y \frac{\partial H}{\partial y}\right) + \frac{\partial}{\partial z}\left(K_z \frac{\partial H}{\partial z}\right) = S \frac{\partial H}{\partial t} + f(x, y, z, t) \quad (1)$$

where K_x , K_y , K_z are the hydraulic conductivity coefficients in the direction of corresponding coordinate axes, $H=H(x, y, z, t)$ is the piezometric level to be found, $f(x, y, z, t)$ is the function characterizing water infiltration and specific yield in a given point, and $S=S(x, y, z, t)$ is the coefficient with variable piezometric level H of elastic storage. The boundary conditions of Equation (1) are given by the initial values of the function $H(x, y, z, t)$ at time t_0 in any point of the area considered. The coefficients K_x , K_y , K_z and S are determined by the constant geological properties of the aquifer at any point, and by the piezometric level H .

Numerical solution of this equation will be based on the finite difference method using space and time discretization. For the considered area of Equation (1), a grid containing N_x , N_y , N_z nodes in the direction of coordinate axes x , y , z , respectively, will be created. The corresponding difference equations resulting from (1) will be applied to every node of the grid and we shall have a system of $N_x \cdot N_y \cdot N_z$ linear equations in the following form:

$$A_{i,j,k} H_{i,j,k-1} + B_{i,j,k} H_{i-1,j,k} + C_{i,j,k} H_{i,j-1,k} + D_{i,j,k} H_{i,j,k} + E_{i,j,k} H_{i,j+1,k} \\ + F_{i,j,k} H_{i+1,j,k} + G_{i,j,k} H_{i,j,k+1} = Q_{i,j,k} \quad (2)$$

where $1 \leq i \leq N_x$, $1 \leq j \leq N_y$, $1 \leq k \leq N_z$. In the equations of this system, the coefficients $A_{i,j,k}$, $B_{i,j,k}$, $C_{i,j,k}$, $D_{i,j,k}$, $E_{i,j,k}$, $F_{i,j,k}$, $G_{i,j,k}$ with the unknowns, and the known values of $Q_{i,j,k}$, are related to time moment t , and the values of H are calculated for the following moment $t+\Delta t$ (for simplicity, subscripts will be omitted in the sequel).

The equations in system (2) can be interpreted on the basis of water balance in blocks into which the investigated real space area can be divided. Note that, because the form of physical environment space blocks need not be rectangular but may be quasi-rectangular, the investigated area may always be interpreted as ideal parallelepipeds composed of identical cubes. Each of these boxes (grid nodes), determined by integer coordinates i , j , k corresponding to coordinate axes x , y , z , has the parameters $A_{i,j,k}$, $B_{i,j,k}$, $C_{i,j,k}$, $E_{i,j,k}$, $F_{i,j,k}$, $G_{i,j,k}$ which characterize water diffusion through all cube walls in time from t to $t+\Delta t$, and by the parameter $H_{i,j,k}$ representing its piezometric level in time $t+\Delta t$. Although the parameters $A_{i,j,k}$, $B_{i,j,k}$, $C_{i,j,k}$, $E_{i,j,k}$, $F_{i,j,k}$, $G_{i,j,k}$ depend upon the value of piezometric level H (in time t), their change in a time interval is slow enough and thus it can be considered that they do not change in every

such interval. In finite difference equations system, the parameters $D_{i,j,k}$ and $Q_{i,j,k}$ are used as well. The first one determines summary diffusion from a block plus block storage change for the moment $t + \Delta t$, and the second one characterizes the sum of block storage in time t and the infiltration value. In matrix form, this equation system is represented by

$$\mathbf{MH} = \mathbf{Q} \quad (3)$$

where \mathbf{M} is symmetric 7-diagonal matrix, as results from the interpretation of the coefficients $A_{i,j,k}$, $B_{i,j,k}$, $C_{i,j,k}$, $D_{i,j,k}$, $E_{i,j,k}$, $F_{i,j,k}$, $G_{i,j,k}$ and Equation (2).

3. The Parallel Strongly Implicit Procedure (SIP) Algorithm

After analyzing several algorithms for solving sparse system of linear equations, we have found two most suitable candidates for solving Equation (3): Strongly Implicit Procedure (SIP) [1], [2] and Conjugate Gradient (CG) [14]. CG is one of the most modern algorithms for solving sparse matrices. It is based on vector and matrix operations, which are easy to parallelize, so the parallel CG can achieve nearly linear speedup. The popularity of CG is gradually increased with many new preconditioning methods, which accelerate the convergence of the solution. However, our experiments show that despite using different preconditioning methods [6], [7], [8], [9], CG needs 2–3 times more computation than SIP. The reason is that SIP is especially designed on the basis of physical properties of liquid for full exploitation of the regularity (nonzero elements are only on 7 diagonals) and the continuity (the difference between values of two neighboring nodes is small) of the sparse matrix \mathbf{M} for efficient solution. Because the efficiency of our parallel version of SIP is very high (see Sec. 6), CG can be faster than SIP only on a very large system. For smaller systems, i.e. PC cluster, SIP is more suitable than CG.

The main idea of this method is in performing fast low-cost approximate LU decomposition of matrix \mathbf{M} into two 4-diagonal triangular matrices (lower matrix \mathbf{L} , and upper matrix \mathbf{U}) in linear time:

$$\mathbf{LU} \approx \mathbf{M} \quad (4a)$$

or

$$\mathbf{LU} = \mathbf{M} + \mathbf{N} \quad (4b)$$

where \mathbf{N} is “close” to zero matrix. By substitution $\mathbf{M} = \mathbf{LU} - \mathbf{N}$ from Equation (4b) to Equation (3), we have:

$$(\mathbf{LU})\mathbf{H} = \mathbf{Q} + \mathbf{NH} \quad (5)$$

Equation (5) will be solved by iteration:

$$(\mathbf{LU})\mathbf{H}^i = \mathbf{Q} + \mathbf{NH}^{i-1} \quad (6)$$

or

$$(\mathbf{LU})(\mathbf{H}^i - \mathbf{H}^{i-1}) = \mathbf{Q} - \mathbf{MH}^{i-1} \quad (7)$$

Letting $\mathbf{X}^i = \mathbf{H}^i - \mathbf{H}^{i-1}$ and $\mathbf{R}^{i-1} = \mathbf{Q} - \mathbf{MH}^{i-1}$, Equation (7) can be solved in two steps:

$$\mathbf{LY}^i = \mathbf{R}^{i-1} \quad (7a)$$

and

$$\mathbf{UX}^i = \mathbf{Y}^i \quad (7b)$$

Because L and U are sparse 4-diagonal triangular matrices, Equations (7a) and (7b) can be solved in linear time. The iteration process (7a) and (7b) will be repeated, until the X^i vector is close enough to zero vector.

In summary, SIP consists of two parts: the approximate LU decomposition (4a) and the iteration process, which involves solving two sparse triangular matrices (7a) and (7b). It is well known that the SIP method convergence rate increases when, instead of one LU decomposition, two different ones are used, where the second LU decomposition differs from the first one by inverted re-numbering order of index values in one of the i, j, k directions.

4. Parallelization of SIP

The matrix in (3) can be very large (10^6 or more), so parallelization of SIP is of great importance. Parallelization does not only speed up the computation, but also solves the problem with memory for containing the matrix and other intermediate values (about 200MB for matrix with 10^6 equations). In order to parallelize SIP, the data dependency of each step of SIP has been investigated in detail. Each step of SIP has the form of 3-dimensional nested loop. In LU decomposition (4a) and solving $LY^i = R^{i-1}$ (7a) the computation has the following form:

```
for i = 1 to  $N_x$ 
  for j = 1 to  $N_y$ 
    for k = 1 to  $N_z$ 
       $Z_{i,j,k} = f(Z_{i-1,j,k}, Z_{i,j-1,k}, Z_{i,j,k-1})$  ;
```

where Z is a set of variables.

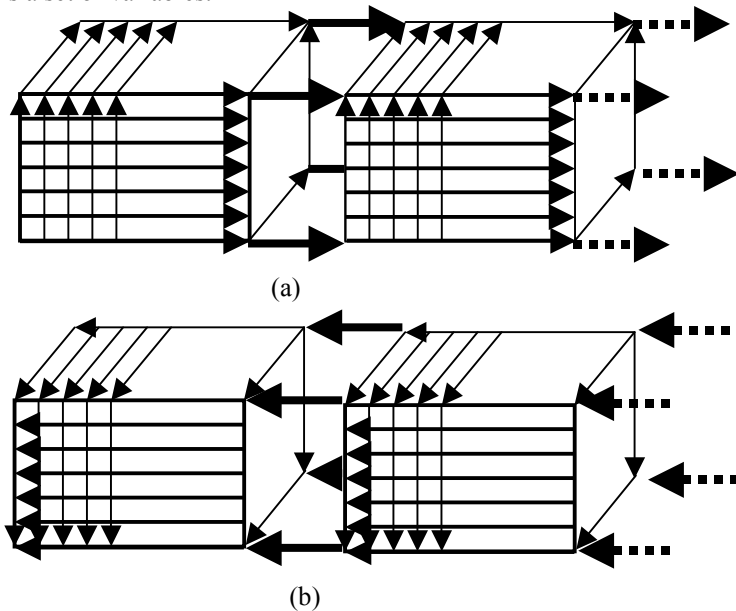


Fig. 1. Parallelizing the Uniform Recurrence Equation.

In solving $UX^i = Y^i$ (7b) the computation has the opposite direction:

```

for i = Nx to 1
  for j = Ny to 1
    for k = Nz to 1
      Zi,j,k = f̂(Zi+1,j,k, Zi,j+1,k, Zi,j,k+1) ;

```

In both cases, the loops have the Uniform Recurrence Equation (URE) form with dependence vectors $d_1=(1,0,0)$, $d_2=(0,1,0)$, $d_3=(0,0,1)$. Different techniques for parallelizing URE are proposed in [10], [11], [12], [13]. In [3], Megson and Chen have analyzed existing methods and proposed a systematic way to parallelize and distribute URE on parallel environment. The data distribution is shown in Fig. 1. The simulation space is divided in the direction of *x-axis*. Each processor computes nodes of the grid in a row, sends the value of the last node to the next processor and continues on the next row. The matrix is distributed among processors, each processor contains only a part of the matrix. The computation behaves as pipeline. We can also create pipeline in *y* or *z* direction. Fig. 1a shows the computation direction of the LU decomposition (4a) and solving $LY^i = R^{i-1}$ (7a). Solving $UX^i = Y^i$ has the opposite direction (Fig. 1b).

If e is the computation time of a node, and c is the communication delay, the speed-up of a p -processor system is:

$$sp = \frac{T_1}{T_p} = \frac{N_x N_y N_z e}{\frac{N_x}{p} N_y N_z e + (p-1)(c + \frac{N_x}{p} e)} \quad (8)$$

where start-up overhead $st=(p-1)(c+N_x e/p)$ is the time when the last processor starts. Equation (8) can be transformed to the form:

$$sp = \frac{p}{1 + \frac{st}{N_x N_y N_z e}} \quad (9)$$

From (9) we can see that the algorithm can reach near optimal speed-up, given that the scale of the simulation space is large enough.

5. Implementation of the Parallel SIP with PVM

When we implemented the parallel SIP with PVM on PC clusters, we found that the time needed to transfer the value of the last node in a row to the next processor is much larger than the time needed to compute values of all nodes in the row. Therefore the communication cannot be totally overlapped by the computation and that the utilization of processors and the performance of the system is decreased in general. Fortunately, the average communication overhead per value can be decreased by grouping data to a large package and sending them at once (Fig. 2). So the algorithm described in Fig. 1 can be modified as follows: each processor computes the values of the nodes in N_{grain} rows and sends N_{grain} values of the last nodes of the rows to the next

processor. By that way, the communication is reduced and can be overlapped by computation, and processors do not have to wait for data. However, grouping data also increases the startup overhead st in (9) and decreases speedup, so N_{grain} should not be large.

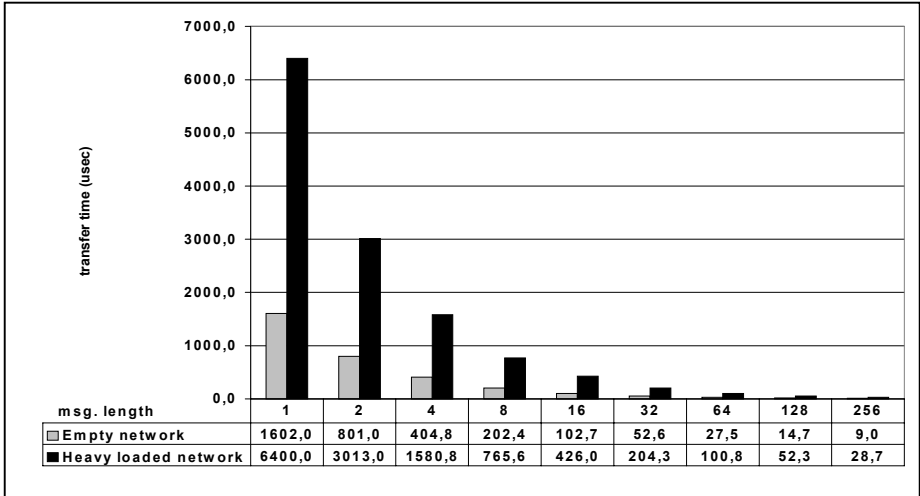


Fig. 2. The dependencies of the transfer time of one floating-point number on the package size.

Fig. 3 shows the dependencies of the time needed for computation and communication on the grain size N_{grain} . The N_{grain} is optimal when the computation time of N_{grain} rows is equal to the time needed for transfer a packet of N_{grain} values, i.e. the intersection of the two lines. Because the communication time in a real computer network is not deterministic, the real optimal grain size should be slightly larger than the theoretical value from Fig. 3 for the worst case.

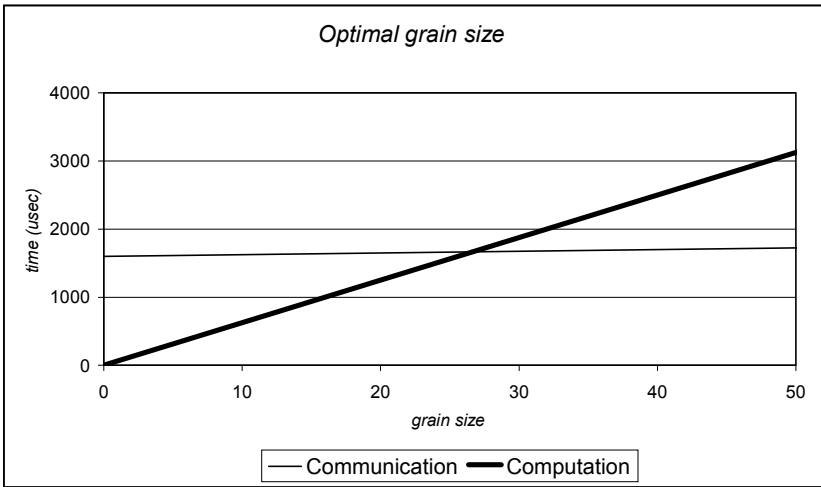


Fig. 3. The optimal grain size for parallel SIP.

6. Experimental Results

Experiments are done on a cluster of 8 Pentium PC connected by 100Mb/s Ethernet. In Fig. 4, we can see the dependencies of the speedup on size of the simulation space and the grain size N_{grain} . The speedup is continually increased with the size of the simulation space. For the largest simulation space of our experiments 640x40x200, the speedup is 7.2 on 8 processors, i.e. the efficiency is 90%. The speedup can be higher for larger simulation space, but the single computer does not have enough physical memory for containing data, so we cannot compare the computation time (the cluster of 8 PC can contain 8 times larger matrix).

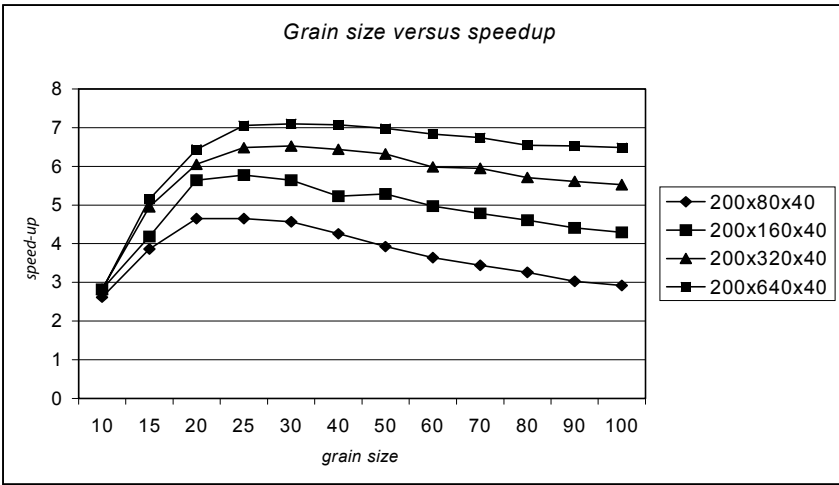


Fig. 4. The dependencies of speedup on grain size and simulation space size.

We also use PG_PVM monitoring tool and ParaGraph visualization tools for analyzing the runtime behavior of the parallel SIP. Fig. 5 shows the communication among processors. We can see the computation behaves as the pipelines described in Fig. 1.

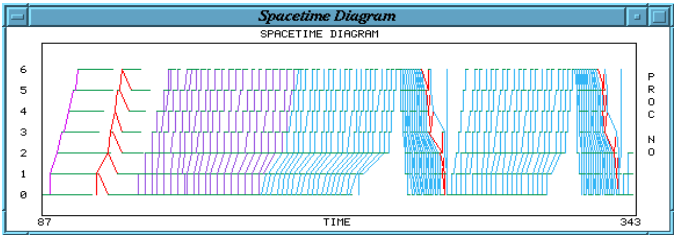


Fig. 5. The communication among processors.

7. Conclusion

We have presented the parallel version of SIP and demonstrated them on numerical solution of the partial differential equation (diffusion type) of the ground water flow model. The parallel SIP is implemented on PC cluster with PVM. As our experiments have shown, the parallel SIP method is very effective and reaches high speedup.

References

- [1] J. A. Alden, R. G. Compton, R. A. W. Dryfe, "Modelling electrode transients: the strongly implicit procedure", *Journal of Applied Electrochemistry*, Vol. 26, 1996, pp. 865–872.
- [2] H. I. Stone, "Iterative solution of implicit approximations of multidimensional partial differential equations", *SIAM J. Numer. Anal.*, Vol. 5., No. 3, September 1968, pp. 530–558.
- [3] G. M. Megson, X. Chen, *Automatic parallelization for a class of regular computations*. World Scientific, 1997.
- [4] L. Hluchý, A. Godlevsky, L. Halada, M. Dobrucký, V. D. Tran, "Ground Water Flow Modeling in Distributed Environment", *Proc. of DAPSYS'98, Workshop on Distributed and Parallel Systems*, Budapest, 1998, pp. 155–161.
- [5] L. Hluchý, A. Godlevsky, L. Halada, M. Dobrucký, "Ground water flow modeling on PC cluster", *ASIS 98 - Advanced Simulation of Systems, XXth International Workshop*, Krnov, 1998, pp. 15–20.
- [6] N. I. M. Gould, J. A. Scott, "Sparse approximate-inverse preconditioners using norm-minimization techniques", *SIAM*, Vol. 19, No. 3, May 1998 pp. 605–627.
- [7] M. Benzi, M. Tuma, "A sparse approximate inverse preconditioner for nonsymmetric linear systems", *SIAM*, Vol. 19, No. 3, May 1998, pp. 968–994.
- [8] E. Chow, Y. Saad, "Approximate inverse preconditioner via sparse-sparse iterations", *SIAM*, Vol. 19, No. 3, May 1998, pp. 995–1023.
- [9] P. Tsompanopoulou, E. Vavalis, "ADI method cubic spline collocation discretizations of elliptic PDEs", *SIAM*, Vol. 19, No. 2, March 1998, pp. 341–363.
- [10] M. Rim, R. Jain, "Valid Transformation: A New Class of Loop Transformations for High-Level Synthesis and Pipelined Scheduling Applications", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 7, No. 4, April 1996, pp. 399–410.
- [11] A. Agarwal, D. A. Kranz, V. Natarajan, "Automatic Partitioning of Parallel Loops and Data Arrays for Distributed Shared-Memory Multiprocessors", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 6, No. 9, September 1995, pp. 943–962.
- [12] A. Darte, Y. Robert, "Constructive Methods for Scheduling Uniform Loop Nets", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 5, No. 8, August 1994, pp. 814–822.
- [13] W. Shang, M. T. O'Keefe, J. A. B. Fortes, "On Loop Transformations for Generalized Cycle Shrinking", *IEEE Trans. on Parallel and Distributed Systems*, Vol. 5, No. 2, February 1994, pp. 193–204.
- [14] A. Basemann, B. Reichel, C. Schelthoff, "Preconditioned CG methods for sparse matrices on massively parallel machines", *Parallel Computing*, vol. 23, 1997, pp. 381–398.

Virtual BUS: A Simple Implementation of an Effortless Networking System Based on PVM

Shinya Ishihara, Seiichiro Tani, and Atsushi Takahara

High-Speed Protocol Processing Research Group
Media Networking Laboratory

NTT Network Innovation Laboratories

1-1 Hikarinooka Yokosuka-Shi Kanagawa 239-0847 Japan

Phone +81 468 59 8897 / Fax +81 468 59 8585

{shinya,tanizo,taka}@exa.onlab.ntt.co.jp

Abstract. This paper describes a simple implementation of an effortless networking system. On the analogy of the conventional computer's bus architecture, we propose the concept "Virtual BUS". In our approach, an application virtually expands its own local bus into the network and accesses various distributed computing resources as if they are on the local bus. We present a light-weight experimental system using the PVM message passing interface and evaluate our approach.

1 Introduction

Recent high-speed and broad-band networking technologies are spurring a trend moving towards effortless networking [1]. An effortless networking environment provides a spontaneous (easy to use, simple) networking process that gives an instant and automatic link to a network, and users are released from bothersome networking. Moreover, in such an environment, the network itself is concealed from users and devices and locations become transparent. Users have a personalized (ubiquitous, nomadic) meta-computing environment wherever they are. Network maintenance and upgrading can be done without worrying about a user's individual communication, because the network is completely separated or hidden from users. There are several approaches to developing such an environment. Jini [2] based on Java and TINA [3] based on CORBA are two of the better known. PVM can also provide a transparent environment through the concept of a parallel virtual machine over heterogeneous platforms. From our experiments with PVM, we expect PVM-based system to be a lighter-weight implementation than one implemented with Jini and TINA.

To clarify the requirements for constructing an effortless networking environment, we designed a model based on the analogy to a conventional computer's bus architecture. In our approach, which we call "*Virtual BUS*", applications virtually expand their own local bus into the network. In other words, they establish a virtual bus in the network and access distributed computing resources, such as memories, cameras and speakers as if they are on a local bus. This

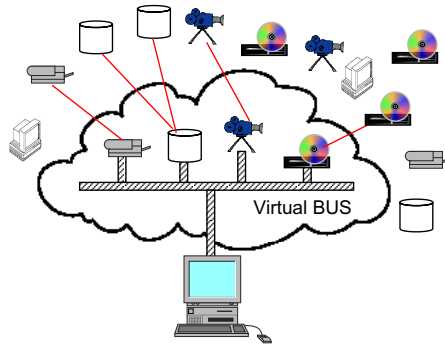


Fig. 1. Conceptual image of Virtual BUS.

model offers a simple way to realize an effortless networking environment. PVM is normally adopted as a platform for high-performance distributed computing. Our approach uses sophisticated PVM control mechanisms to handle distributed tasks and its message passing interface as a high-level communication library. In PVM, several platforms (almost all vendor's Unix and Win32) are supported and libraries for high-level languages such as C/C++ and Java are provided. These features encouraged us to develop a PVM-based prototype system.

Section 2 describes the Virtual BUS concept and its requirements. Section 3 describes the design and implementation of the Virtual BUS in PVM. Section 4 presents our experimental system and a discussion. Finally, some conclusions are given in section 5.

2 Virtual BUS and Its Requirements

Virtual BUS is a simple model for effortless networking service in the distributed environment. Figure 1 shows a conceptual image of Virtual BUS.

In Virtual BUS, the organization of distributed computing resources is modeled as the bus of a computer system. Users construct their own virtual bus in a network and the virtual bus connects the various distributed computing resources available in the network. In realizing such an environment, there are two key issues: uniformity and virtuality.

Uniformity is realized by preparing a uniform interface for local bus access. A bus is a common communication channel to its participants, so it can be regarded as a proxy service of communication between applications and resources. The proxy mechanism that acts bus attach point enables us to establish a virtual bus in the network. Of course, a bus interface for applications and resources is needed. We should define a uniform resource access method as opposed to a resource-specific one.

Virtuality is supported by preparing a resource look-up mechanism. In the case of a legacy network, such as the Internet, a location identifier (address)

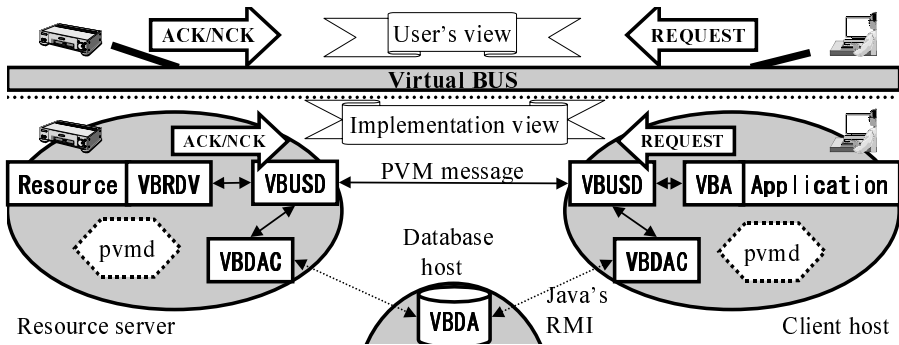


Fig. 2. Virtual BUS architecture.

and resource identifier (port number) are needed to identify specific computing resources. For Virtual BUS, a richly functional directory service is needed. In order to give more freedom to users, the environment chooses suitable resources in compliance with user's demand. A user should be able to request some resources with their characteristics rather than have to specify an individual resource's exact location such as address and port number.

3 Design and Implementation

To satisfy the requirements described above, we designed a simple Virtual BUS architecture, which is shown in Figure 2. The architecture consists of five functional modules, which are realized as PVM tasks.

VBUSD: A daemon process with the role of the attach point for applications and resources.

VBA: Software adaptor that acts as the bus interface for applications.

VBRDV: Software adaptor that acts the bus interface for resources.

VBDA: Directory service that manages resource information.

VBDAC: An interface to directory service.

3.1 Virtual BUS Attach Point

As the virtual bus attach point for applications and resources, the VBUSD, Virtual BUS Daemon, is deployed on each host. Cooperation among VBUSDs establishes the virtual bus in the network. The VBUSD provides the proxy service of communication between applications and resources and realizes transparency of location and distribution. This function is achieved with a connection management table, which manages which application opens which virtual bus and attaches which resource (remote or local). In our implementation, applications and resources are identified by its PVM task IDs, so message proxying are easily managed by VBUSDs.

```

#include <vba.h>                                // Access methods definitions

main(){
    const char ResType[]="ENCODER";    // Type of the required resource
    int ParamCnt=2;                    // Number of attributes
    String Params[2]={String("format=mpeg"),    // Attributes of the
                      String("contents=good movie")}; // required resource
    VBA vba;                            // Creation of Virtual BUS interface
    int vbid=vba.open();                // Constructing its own virtual bus
    String ResName=vba.get(vbid,ResType,    // Attaching the resource
                           ParamCnt,Params); // to the bus
    vba.write(vbid,ResName,addr,data); // Writing some control commands
    vba.read(vbid,ResName,addr,data); // Reading status of the resource
    vba.release(vbid,ResName,           // Detaching the resource
                ParamCnt,Params);      // from the bus
    vba.close(vbid);                   // Destructing its own virtual bus
}

```

Fig. 3. A sample code of a Virtual BUS application.

3.2 Virtual BUS Interface

The Virtual BUS interfaces for applications and resources are software adaptors including the PVM library. The interfaces are provided as C/C++ libraries. Developers can make Virtual BUS applications and resources with interface libraries without engaging in bothersome networking. Applications and resources constructed using the Virtual BUS interface libraries are launched as PVM tasks and identified by PVM task IDs.

VBA, the Virtual BUS interface for applications, provides uniform resource access methods. Figure 3 shows a sample code of Virtual BUS application using VBA library. The requests (OPEN, GET, WRITE, READ, RELEASE, CLOSE) are translated into PVM messages, then the messages are sent to VBUSD and acknowledgments are received.

VBRDV is the Virtual BUS interface for resources and provides the functions of resource characteristic registration and response against application's requests. Resources with VBRDV are accessed as memory mapped I/O devices. Figure 4 shows a sample code of Virtual BUS resource using VBRDV library.

In the implementation of VBA and VBRDV, we hide PVM related processing in our macro method (`write()`, `read()`, `...`). The message queueing mechanism of PVM is also hide by `deq()` call. So users can easily develop their applications and resources without considering parallel computation environment and the knowledge of PVM.

```

#include <vbrdv.h>                                // Definitions for resources

main(){
    const char ResType[]="ENCODER";              // Declaration of resource type
    int ParamCnt=2;                               // Number of attributes
    String Params[2]={String("format=mpeg"),      // Declaration of
                      String("contents=good movie")}; // attributes
    const int NumberOfUser=1;                     // Number of exclusive users
    const String ExpireTime("1999:12:31:23");     // Expiration time
    VBRDV drv(ResType,ParamCnt,Params,           // Creation of interface and
              NumberOfUser,ExpireTime);          // registration
    while(1){                                     // Main loop
        while(drv.empty()==FALSE){               // Checking a request
            VbrdvContainer vcon;                 // Preparing a request receiver
            vcon = drv.deq();                     // Dequeuing a request
            switch(vcon.RequestCode){             // Branch by a kind of request
                case VBRDVWRT:                    // Process for writing received
                    ...                           // data to specified address
                    drv.Notify(VBRDVWRTACK);      // Notification of
                    break;                        // an acknowledgment
                case VBRDVRD:                     // Process for reading data
                    ...                           // from specified address
                    drv.Notify(VBRDVRDACK,data);  // Notification of
                    break;                        // an acknowledgment
            }
        }
    }
}

```

Fig. 4. A sample code of a Virtual BUS resource.

3.3 Resource Look-Up Mechanism

Referring to Service Location Protocol [4], we designed the resource look-up mechanism, VBDA, as a daemon process of directory service based on Java. VBDA manages resource information, i.e. the resource ID, VBUSD ID, resource type, attribute, and expiration time. The resource ID and VBUSD ID are, in fact, PVM task IDs. The resource type represents a resource's basic functions, for example, "ENCODER", "ROBOT", or "MEMORY". The attribute represents additional information, for example, "format=mpeg", "contents=mt. fuji", or "rate=6Mbps". Expiration time is useful for deleting obsolete entries.

The interface to the directory service, VBDAC, accepts resource look-up requests from VBUSDs with PVM message passing interface by jPVM and accesses VBDA using Java's Remote Method Invocation (RMI). VBDAC is the bridge between PVM and Java environment.

3.4 Typical Life of Virtual BUS Applications and Resources

We explain the behavior of Virtual BUS applications and resources in a video-on-demand (VOD) system as an example. Let's consider the situation where a user wants to watch "good movie" served on an MPEG stream.

1. A MPEG encoding resource and a MPEG decoding resource are launched. They request registration of their characteristics to VBUSD on the same host. VBUSD registers this information to the resource directory in VBDA through the interface of directory VBDAC.
2. A user starts the application of VOD. The VOD application requests for opening its own virtual bus to VBUSD using the OPEN request of VBA. VBUSD makes a new entry for the VOD application on its connection management table, and the virtual bus for the application is constructed.
3. Then the application requests a decoding resource be attached to the bus using the GET request of VBA. VBUSD accesses VBDA through VBDAC and looks up directory entries with specified resource type "DECODER" and specified attributes such as "format=mpeg" and "host=\${the application's host}". After successfully looking up the entries, the GET request is sent directly to the resource. If the decoding resource is available, it acknowledges with an ACK message, otherwise a NCK message is sent.
4. The application also gets an encoding resource. Resource look-up is done with a specified resource-type "ENCODER" and specified attributes "format=mpeg" and "contents=good movie". VBUSD sends a remote GET request to another VBUSD that manages the encoding resource when the chosen encoding resource is not managed by the same VBUSD used for the VOD application. The other sequences are the same as in decoding resource's case.
5. Once resources are attached, the application can communicate to them using the WRITE and READ requests. Control commands for attached resources are executed by writing some data to appropriate memory address of the resources. The status of attached resources is checked by reading from appropriate memory address of the resources.
6. When the application finishes, attached resources are released from the bus by sending the RELEASE requests to attached resources through VBUSD. Finally, the application closes its virtual bus using the CLOSE request.

4 Experimental System and Discussions

We built an experimental system on multi platforms (Solaris 2.6, FreeBSD 2.2.8 and Windows NT 4.0 Service Pack 3) connected by ATM and a 100M-Ethernet. For Unix environment, we used GNU C++ 2.8.1 and PVM 3.4 beta 6. For Windows environment, we adopted Visual C++ 5.0/6.0 and Win32 port version of PVM[5] with Ataman's rshd[6]. Java based resource look-up mechanism was made by jdk 1.1.6[7]. jPVM 1.1.4[8] played an role of a bridge between PVM and Java environment.

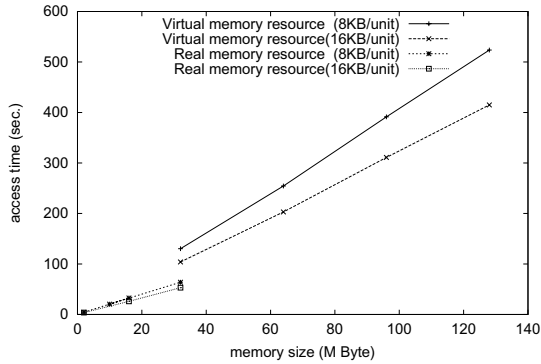


Fig. 5. Performance of writing/reading to/from memory resources.

4.1 Experimental Results

We developed some Virtual BUS applications and evaluated our system. For VOD service, we developed MPEG2 and Motion-JPEG encoding/decoding resource drivers, and a control application for them. We composite different format encoding resources and developed a formatless encoder proxy. The remote observation system is composed of a camera control application and camera resource driver with VOD's encoding/decoding resource drivers. A network robotics control system was implemented by integrating a miniature robot resource driver and control application with the remote observation system.

We also developed real and virtual memory resources. The virtual memory resource is a memory proxy that gathers many real memory resources and attaches on its own Virtual BUS then acts as a huge memory for a memory user. Figure 5 shows the performance of the memory resources. In this experiment, 64, 96, 128M byte virtual memory resources are constructed using 32M byte memory resource. The Y-axis shows the elapsed time for writing/reading data to/from a virtual memory resource with accessing 8k/16k byte block units and the X-axis shows the size of a memory resource. The access time for a virtual memory resource is almost twice as slow as that of real memory resource in the current implementation.

4.2 Discussions

We think PVM is easy enough to use and powerful enough to develop such systems. Unfortunately, with PVM, a different virtual machine is constructed for each user. This limitation means a person's PVM-based system can be used by only him or her. Supposing virtual user for PVM-based middle-ware would overcome this limitation. In the present implementation, stream data is sent over the ATM layer rather than through Virtual BUS. We expect PVM over ATM

[9] will improve the performance of Virtual BUS. For further improvement of the performance, VBUSD may be designed by extending pvmd with directory access and connection management like in Kemelmakher's approach[10].

For a wide area network, it is important to secure scalability and support QoS and security. We have one idea to apply hardware acceleration. We will try to embed Virtual BUS functionality into ATTRACTOR, which is our telecommunication-oriented reconfigurable hardware [11]. This would effectively support the required QoS for real-time traffic monitoring, resource roaming, dynamic network reconfiguration and so on.

5 Conclusions

This paper presented a simple PVM-based implementation of an effortless networking system, which is called Virtual BUS. In the environment, we can access any distributed computing resources in computer networks as mapped I/O devices, as if they are attached to a conventional local bus in a PC. We constructed an experimental system utilizing many PCs and peripherals, and then demonstrated some applications to show the feasibility of our concept.

References

1. J. Callahan, "Moving Toward Effortless Networking," *IEEE Computer Magazine*, vol. 31, pp. 12–14, Nov. 1998.
2. SUN Microsystems, "Jini(tm) Connection Technology." <http://www.sun.com/jini/>.
3. TINA Consortium, "TINA-C Home Page." <http://www.tinac.org/>.
4. J. Veizades, E. Guttman, C. Perkins, and S. Kaplan, *Service Location Protocol*, June 1997. RFC 2165.
5. Oak Ridge National Laboratory, "PVM WIN32 PORT." <http://www.epm.ornl.gov/pvm/NTport.html>.
6. Ataman Software, Inc., "Ataman Software, Inc. Home Page." <http://www.ataman.com/>.
7. SUN Microsystems, "Java(tm) Technology Home Page." <http://java.sun.com/>.
8. "jPVM: A native methods interface to PVM for the Java platform." <http://www.isye.gatech.edu/chmsr/jPVM/>.
9. M. Lin, "Release Note of PVM-ATM 3.3.2.0." <ftp://ftp.cs.umn.edu/users/du/pvm-atm/www.html>.
10. M. Kemelmakher and O. Kremien, "Scalable and Adaptive Resource Sharing in PVM," in *5th European PVM/MPI User's Group Meeting*, vol. 1497 of *Lecture Notes in Computer Science*, pp. 196–205, Springer-Verlag, Sept. 1998.
11. T. Miyazaki, K. Shirakawa, M. katayama, T. Murooka, and A. Takahara, "A transmutable telecom system," in *Proc. 8th International Workshop on Field Programmable Logic and Application*, vol. 1482 of *Lecture Notes in Computer Science*, pp. 366–375, Springer-Verlag, 1998.

Collective Communication on Dedicated Clusters of Workstations

Lars Paul Huse

Scali AS, Hvamstubben 17, N-2013 Skjetten, Norway
lph@scali.com <http://www.scali.com>

Abstract. Fast scalable collective operations are very important for achieving good application performance for highly parallel systems. This paper discusses scalability and absolute performance of various algorithms for collective communication. Performance is measured on the world largest SCI cluster, a machine with 96 dual Pentium II nodes interconnected with SCI, showing good scalability and performance.

1 Introduction

MPI (Message Passing Interface) [9] is a well established communication standard. The collective communication primitives in MPI cover most common global data movement and synchronization primitives. ScaMPI [13] is Scalit's high performance implementation of MPI. ScaMPI currently runs over local and SCI (Scalable Coherent Interface) [7] shared memory on Solaris, NT and Linux for SPARC and x86 based workstations. SCI is a standardized high-speed interconnect based on shared memory, with the nodes connected in closed rings.

In this paper we look at algorithms and achieved performance for various implementations of collective operations executed on a large dedicated cluster of workstations [11]. The paper starts with an overview of ScaMPI and the machine used in the experiment. The different basic algorithms are then discussed, and their usage in the different collective operations is described and performance measured.

MPICH [10] is a free wide-spread high performance MPI implementation, and is used to compare the performance of ScaMPI. Unfortunately no good *channel interface* implementation exists for MPICH over SCI yet. Since collective operations in MPICH is based on `MPI_Send()` and `MPI_Recv()`, the MPICH algorithms use ScaMPI's basic send and receive to get a fair comparison.

2 ScaMPI for Local and SCI Shared Memory

ScaMPI [13] uses a write only protocol, over the shared memory segments forming unidirectional communication channels between sender and receiver. For best performance the SCI memory segments reside on the receiver side. ScaMPI uses three types of buffers all residing in shared memory. These are *channel*, *eager* and *transporter* buffers.

A channel buffer is a ring buffer for send request and status. ScaMPI uses two channel buffers for each communicator, one for collective operations and one for other traffic, to avoid resource conflicts. Messages up to 32 bytes are piggy-backed to the header, and transferred directly in a channel buffer entry. A sequence number and checksum protect against data hazards caused by the data reordering that may occur in caches, IO bridge or network, making the transfer self synchronizing and reducing protocol overhead to a minimum.

The eager-buffers are buffered connections, filled on a first-come-first-served basis with one buffer per message. The sender first copies the user data to a free eager-buffer and then notifies the receiver using a channel buffer entry. The receiver then copies the data to the user buffer, and releases the buffer for reuse.

The transporter buffers are used for large transfers, splitting the message over one or more buffers. The sender first make a send request through a channel buffer entry. The transfer starts only after a matching receive has been posted (rendezvous-protocol), with the receiver setting the pace for the transfer.

A device driver performs creation/destruction and map/unmap of the SCI shared memory segments. The SCI network guarantees correctness if data is delivered [7]. Data delivery is ensured by check pointing the driver state before and after all copying to SCI shared memory, and retransmitting if the transfer is incomplete. The data movement is done using high-speed memory copy, (*64 byte copy* for SPARC and *MMX copy* for x86). All data copy is done in user mode, thus if the interconnect is operational the OS is not called during data exchange.

3 Collective Data Movements

A collective operation involves all N processes in a communicator, e.g., all processes in the program, where $N = P + Q$ and $P = 2^p > Q \geq 0$. All processes have a unique rank r between 0 and $N-1$. [2] define 4 basic building blocks for collective communication; broadcast, combine-to-one, scatter and gather. Since these operations are similar, various implementations of broadcast are in the rest of the section detailed to illustrate collective data movement. Before the broadcast all data reside in the *root* process, and are then migrated to all processes. To ease explanation process 0 is assumed to be the root.

When using a *sequential* or *chain tree*, broadcast is performed in $N-1$ steps. In a *binomial tree* the number of active processes doubles in every step, hence a broadcast will use $Ln = \lfloor \log 2(N-1) \rfloor + 1$ steps ($Ln \leq N$).

Sequential Tree: (One-to-all for scatter/All-to-one for gather) The conceptually easiest way to broadcast data is for the root process to send to all other $N-1$ processes, while all other processes receive from the root.

Chain Tree: Process 0 sends to process 1 and process $N-1$ receives from $N-2$. Processes $r \in [1 \dots N-2]$ receive from $r-1$ and send to $r+1$. The pipelined nature of the algorithm gives successive operations high throughput.

Distance Power-of-Two Binomial Tree: In $s \in [1 \dots Ln]$ steps all process r with data sends to process $r_x = r + 2^{Ln-s-1}$ if $r_x < N$. For non power of two N the average spread time is higher than other binomial trees.

Divide-by-Two Binomial Tree: In $s \in [1 \dots Ln]$ steps process 0 in all groups send to $r_x = \lfloor (2 + \max r)/2 \rfloor$ which receive from 0. All groups with more than two processes are then split in $\Phi = [0 \dots r_x - 1]$ and $\Psi = [r_x \dots \max r]$ and new ranks $r' = r - r_x$ assigned to Ψ .

Power-of-Two Remainder Binomial Tree: A group of P processes is formed by excluding Q processes, e.g., with odd rank $< 2Q$. Since the P group has power of two members, a perfectly balanced binomial spread can be performed. The Q remaining processes then receive data from Q processes inside the P group, e.g., with even rank $< 2Q$.

Binary Tree: Each node but the root receive from one node, and all sends to up to two other nodes, using $2 * \lfloor \log 2(N + 1) \rfloor - 1.5 \pm 0.5$ steps to complete, i.e., a near doubling of steps compared to a binomial tree.

4 Experiment Setup

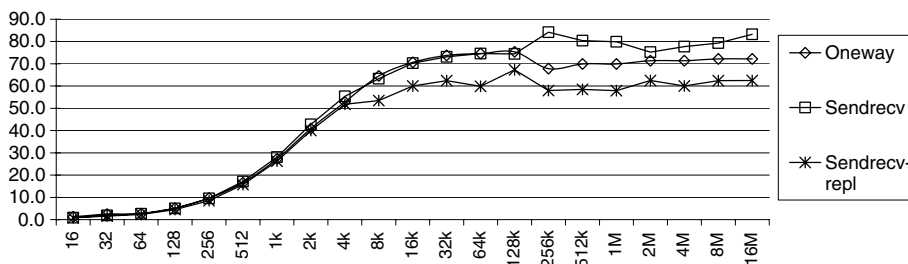


Fig. 1. Point-to-point bandwidth [MB/s] as function of message length [Bytes].

The benchmarked machine [11] consists of 96 standard dual 450 MHz P-II based PCs (440GX chipset) with 512 MB memory and Solaris 2.6, interconnected with PCI-SCI cards from Dolphin [6]. The PC internal PCI bus operates on 33 MHz giving a gross bandwidth of 133 MB/s, of which the SCI card can utilize 90 MB/s. Since each SCI card contains two SCI link controllers, they act as a distributed switch over the internal B-link bus, connecting the nodes in a 12x8 2D torus. On each SCI-link half of the gross bandwidth of 500 MB/s can be used for payload traffic [5], giving each node 70 MB/s on the 8 node ringlets (45 MB/s on the 12 node ringlets). MPI ping latency is 9.2 μ s between nodes sharing SCI ringlet and otherwise 10.3 μ s. Point to point network bandwidth (Fig. 1) is little affected by the nodes relative placement in the network. SMP internal performance is 4.5 μ s and 130 MB/s. The machine has an aggregated bandwidth of up to 3.9 GB/s available to user applications.

Due to lack of priority of SCI response messages in the link-controller [6], an internal livelock may occur under high traffic. This is detected by timers and resolved by the device driver, but introduces unproductive timeslots. Since the livelock is traffic dependent, dividing the communication into phases with reduced

connectivity separated by barriers [4] may therefore improve performance for large configurations. An example of this will be given later for `MPI_Alltoall()`. Checking SCI buffer availability before sending data to the adapter generally reduces performance, but improves livelock recovery (early detection and backoff). ScaMPI therefore dynamically uses this internally, transparent to the user.

5 Performance of the Basic MPI Collective Operations

MPI's API and functional behavior of collective operations are detailed in the standard [9], but only advice is given concerning implementation. In this section a wide set of algorithms; implementation and performance are discussed.

MPI data types may be non-contiguous, thus sending data to oneself can't solemnly be replaced by a memory copy. In most cases MPICH solves this by using an immediate send and a blocking receive. Using this scheme for ScaMPI would result in at least two memory copies and possibly two context switches. ScaMPI has therefore solved this by a memory copy if the data are contiguous, and otherwise by using `MPI_Pack()` and `MPI_Unpack()` to a temporal buffer.

For time measurement the CPU internal cycle counter is used. Since reading the cycle counter (the `RDTSC` instruction) and storing it is as low as ≈ 90 ns, each individual collective call is timed. To reduce startup effects (caching, setup, connection management etc.) the first measurement in all series are discharged.

6 Barrier

Barriers are used for time synchronization between processes. A barrier carries no data payload, and is therefore only dependent of latency. The simplest way to implement a barrier is to send and receive a zero byte message with all other processes, in an alltoall style. A barrier can also be implemented as a gather followed by a broadcast. MPICH 1.0.x uses this approach, with process 0 sequentially receiving then sending to all other processes. Barriers can alternatively be implemented as distributed binomial trees, where for $s \in [1 \dots Ln]$ steps all processes r sends to $(r + 2^{s-1} \bmod N)$ and receives from $(r - 2^{s-1} \bmod N)$. After p steps each process is synchronized with $\min(2^p, N)$ processes. This algorithm requires only half the steps of a binomial gather-broadcast approach. MPICH 1.1.x uses a variant of this with a power-of-two-remainder split which reduces traffic, but gives an extra step if N is not power of two.

ScaMPI is based on the shared memory paradigm with direct access from user space. A barrier has therefore been implemented with direct writing to remote mapped memory and reading (spin-locking on) local memory in a gather-broadcast approach. To enable this ScaMPI allocates memory segments forming octo-trees (generalized binary trees with fanout of 8) for each communicator.

As table 1 shows; a binomial tree based gather-broadcast based barrier achieves good performance, but distributed binomial trees perform better. The SHM based barrier have better timing than a single `MPI_Send()` / `MPI_Recv()` for up to 8 nodes (1.3 μ s for two SMP internal processes)! On barriers ScaMPI

Table 1. Performance for different barrier implementations over SCI[Šs].

Nodes	Basis	2	8	32	48	64	96
All send and receive	MPI	25.0	178.2	877.6	1373.1	1902.9	2816.6
Sequential trees	MPI	26.2	130.0	659.3	1041.3	1425.0	2192.4
Binomial: split in 2	MPI	26.4	81.6	137.1	184.3	174.7	305.2
Binomial: power of 2	MPI	26.5	81.1	137.9	158.1	173.2	221.3
MPICH power two split	MPI	24.6	75.2	128.3	165.4	165.4	255.8
Distributed binomial	MPI	19.4	59.5	105.4	129.3	129.8	153.1
Fast barrier	SHM	8.1	9.3	24.3	26.2	29.7	33.5
Cray Origin 2000	?	25.9	107.1	510.6	434.6	739.7	-

outperforms a far more expensive 128 processor state-of-the-art Cray Origin 2000 [8].

7 Broadcast

Broadcasts are used to scatter equal data from the root process to all other processes. The simplest implementation of broadcast use sequential or chained tree, giving a high latency. Since one node can't saturate the SCI network, it is a good idea for the processes to redistribute its received data to others, e.g., as a binomial tree. MPICH uses a power-of-two binomial tree approach, while the divide-by-two binomial tree is the current ScaMPI implementation.

Throughput of broadcast for a 96 node configuration was within $\pm 10\%$ of the theoretically expected bandwidth, Bw^1 for sequential tree and $(N-1)Bw/Ln$ for binomial tree. The chain tree didn't make $(N-1)Bw$ for successive broadcasts, but still an impressive $\approx 70 \times Bw$.

8 Reduce and Allreduce

Reduce uses an operator to combine data from all processes and places the final result in the root process. Examples of pre defined operations are sum, product, and, or, xor, max and min. Allreduce is a reduce, where all processes get the result. A binomial tree can be used to speed up the operations, e.g., as done by MPICH. For non zero root reducing with a commutative² operator can be done by remapping rank r to $r' = ((r - \text{root}) \bmod N)$. For non-commutative operators the result can be reduced to rank zero (keeping the ordering) and then transferred to the root process.

The simplest way to implement an allreduce is to perform reduce in a virtual root, e.g. 0, followed by a broadcast. If N is power of two allreduce can be computed in $s \in [1 \dots Ln]$ steps where all processes r exchange and reduce

¹ Bw = point-to-point bandwidth of Fig. 1.

² Commutative operator: $a \text{ op } b == b \text{ op } a$.

the accumulated result with $r_x = r \text{ xor } 2^{s-1}$, similar to the data movement butterflies of an FFT (Fast Fourier Transform). The algorithm can handle non-commutative operators by sorting the operator input based on r relative to r_x . Rabenseifner [12] has made an improved generalization of this using a power-of-two-remainder split, to which we made minor adaptations. If N is power of two a distributed binomial tree as described for barrier, can be used for commutative operators. This approach can easily be generalized for all group sizes.

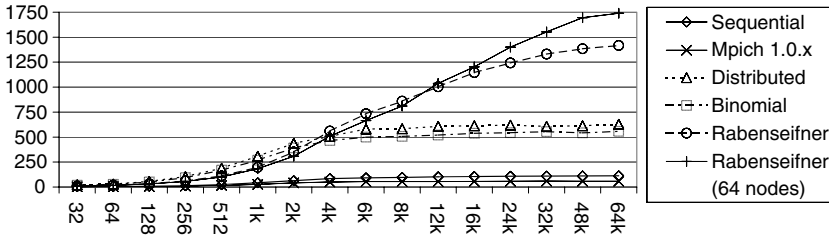


Fig. 2. Allreduce performance [MB/s] for MPI_MAX with MPI_INT on 96 nodes.

As can be seen in Fig. 2 the binomial tree has the expected $(N-1)/Ln$ performance improvement over the sequential tree. The distributed binomial tree gives better performance, while Rabenseifner's algorithm more than double performance for large vectors due to overlap in communication and calculation.

9 Alltoall

In alltoall communication all processes exchange unique chunks of data with all other processes. MPICH implements this by starting all (immediate) sends and receives in parallel and wait for all to finish. Since ScaMPI uses queues and dedicated threads for immediate call [13] (separate for send and receive), this approach is resource demanding for large groups. A more pipelined approach is to send each chunk as a series of sub-chunks with size less than the eager buffer size. The sub-chunks can be sent in series of one or more to one or more processes at the time before receiving, making use of the buffering in the eager communication mode. Since a single two-way data exchange can keep the PCI bus close to saturated, pairing senders and receivers may improve performance. One way of making this is a circular neighbor approach, where in $s \in [1 \dots N-1]$ steps all processes r send to $(r + s \bmod N)$ and receive from $(r - s \bmod N)$.

The connectivity can be reduced by virtually forming an $X \times Y$ 2D mesh with only communication along the two axis, i.e., if two processes are not virtual neighbors they have to communicate through an intermediate process. Gathering all data from a sender to its receiver to one transfer will increase message size, and thereby possibly improve performance (Fig. 1). Since $(X-1)/X$ of the data has to be transferred twice, this approach is inefficient for large chunks.

As observed by [4] considerable time skew will occur if the processes are not synchronized. Processes lagging behind in the communication process will have a hard time sending their data because of the increased amount of incoming data, leaving them even further behind and increasing the chance of the PCI-SCI livelock. Introducing barriers to separate traffic in different steps will reduce this skew, but demands a fast barrier implementation to be efficient. As previously shown ScaMPI has a very fast barrier implementation.

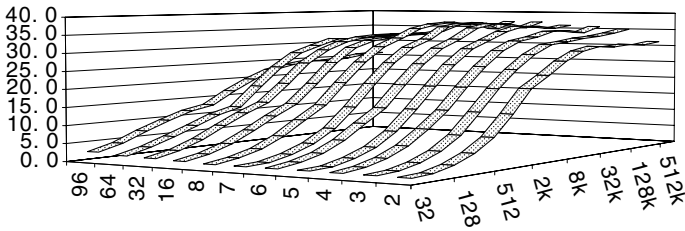


Fig. 3. Bandwidth [MB/s] of alltoall as function of chunk size [Bytes] and nodes.

Alltoall peak performance for two SMP local processes is 54.6 MB/s. Fig. 3 shows the alltoall SCI network performance. The circular neighbor approach usually gives the best performance, but for large chunks on small groups (≤ 8) a slightly modified MPICH approach is better. This is the reason for the apparent performance drop from 8 to 16 nodes and not the SCI network. Barrier synchronization between each step improves performance up to 50% for small chunk sizes (≤ 4 KB), but has only minor effects for large chunks. The virtual 2D-torus approach didn't make it to the performance top due to usage of temporary buffer. The scaling for our machine is better than on a Cray Origin 2000, which at peak delivered 42.1 MB/s for 2 processes, 26.9 MB/s for 16 processes and ends up with a mere 7.3 MB/s for 32 processes[8].

10 Conclusion

MPICH has done a performance boost when going from 1.0.x to 1.1.x for collective operations. The ScaMPI algorithms have comparable or better performance than MPICH 1.1.x algorithm for SCI based clusters. Lowering connectivity by 'barrier insertion' reduces timing deviation. Since transfer time increases with message size, the effect is best for small messages.

Good absolute performance for collective operation on large SCI based clusters has been shown. By selecting algorithms suitable for large networks we have shown collective latency (barrier performance) of $const * \log(N)$ and near constant bandwidth per node for alltoall communication. These are good criteria's of scalable systems, and justify dedicated clusters of workstation for high-performance computing.

11 Related and Further Work

Several projects exist to make effective implementations of MPI. Argonne National Laboratory and Mississippi State University initially made MPICH and Ohio Super-computer Center made LAM, which are free implementations. Several MPI implementations use MPICH with their own low-level driver, e.g., MPICH-PM/CLUMP from Real World Computing Partnership. Some other machine vendors have made proprietary MPI implementation; e.g., CRI/EPCC MPI for Cray T3D which is project at SGI/Cray Research and Edinburgh Parallel Computing Center.

There is also research effort of making performance effective collective communication based on the LogGP [1] model. [3] use an α -tree similar to the divide-by-two binomial tree, but with multiply-by- α ($0.5 \leq \alpha \leq 1/N$) for group splitting, thus selecting the data exchange partners based on latency and bandwidth. Improvements can still be made to enhance ScaMPI collective performance for small messages. [2] proposes alltoall bucket-collect approach with only Ln steps. Binomial or binary tree implementation of scatter/gather/all-gather to make messages longer (see Fig. 1) may also improve performance.

Acknowledgment. A special thanks to service team at PC2 in Paderborn and Thierry Matthey at Parallab for assistance with the measurements.

References

1. Alexandro A., Ionescu M., Schauser K.E., Scheiman C.: LogGP: Incorporating Long Messages into the LogP Model - One Step Closer towards a Realistic Model of Parallel Computation. Proc. 7th Annual ACM Symposium Parallel Algorithms and Architectures (1995).
2. Barnett M., Gupta S., Payne D., Shuler L., van de Geijn R., Watts J.: Interprocessor Collective Communication Library. Proc. of Supercomputing '94 (1994).
3. Bernaschi M., Iannello G., Lauria M.: Experimental Results about two MPI Collective Communication Operations. Proc. 7th International Conference on High Performance Computing and Networking Europe (1999).
4. Brewer E.A., Kuszmaul B.C.: How to Get Good Performance from the CM-5 Data Network. Proc. of the 8th International Parallel Processing Symposium (1994).
5. Bugge H.: Affordable Scalability using Multicubes. Proc. of SCI Europe '98 (1998)
6. Dolphin ICS: PCI-SCI Bridge Functional Specification. Version 3.01 (1996).
7. IEEE standard for Scalable Coherent Interface IEEE Std 1596-1992 (1993)
8. Thierry Matthey - Parallab: Personal communication (1999)
9. Message Passing Interface Forum: MPI: A Message-Passing Interface Standard. Version 1.1 (1995)
10. MPICH: Portable MPI Model Implementation. Version 1.1.1 (1998)
11. PC2 Scalable Compute Server: <http://www.uni-paderborn.de/pc2/systems/psc>
12. R. Rabenseifner: A new optimized MPI reduce algorithm
http://www.hlr.de/structure/support/parallel_computing/models/mpl/myreduce.html (1997).
13. ScaMPI users guide Version 1.6.0 - April 1999. <http://www.scali.com>

Experiences Deploying a Distributed Parallel Processing Environment over a Broadband Multiservice Network

Javier Corbacho-Lozano, Oscar-Iván Lepe-Aldama¹, Josep Solé-Pareta, Jordi Domingo-Pascual

{corbacho, oscar, pareta,jordid}@ac.upc.es
Universitat Politècnica de Catalunya, Spain

Abstract. This paper addresses part of the work that is being carried out within the SABA project. We are deploying and using a virtual parallel machine over a network-of-workstations, which communicates by means of an ATM based broadband multiservice network. The objective is to asses, by experimentation, the effect on the performance of this distributed parallel processing environment produce by the selection of the networking technology and the level of background traffic crossing the network.

1 Introduction

This paper addresses part of the work that is being carried out within the SABA (New Services for the Broadband Academic Network) project. SABA is embedded within the telematic services and applications research program supported by the Spanish government. The project focuses on the development and evaluation of new proposals of technologies, architecture and protocols for communications networks. Moreover, the project considers the use of this technology for various application environments; such as, computer supported collaborative work, videoconferencing and multimedia services, and distributed processing.

Within SABA, we are deploying and using a virtual parallel machine (VPM) over a network-of-workstations (NOW), which communicate by means of an ATM based broadband multiservice network. The objective is to asses, by experimentation, the performance achieved by this distributed parallel processing environment (DPP-E) both, when deployed over a local area and when deployed over a wide area. This is important because, even though the deployment of ATM based NOW's is becoming common ground and there are analytical predictions of its use as a DPP-E under ideal conditions [1], there are no reports² of experimental observations using ATM based NOW's for DPP-E under less idealistic conditions. This paper only describes experiences with the DPP-E over a local area ATM network.

¹ Oscar-Iván Lepe is assistant researcher at Centro de Investigación Científica y Educación Superior de Ensenada, México. He is currently at UPC with a Ph.D. grant.

² Related reports we found either use a non ATM network to support their PVM based DPP-E [2] or do not use full DPP-E like ours over ATM [3, 4].

It is well known that the performance of a DPP-E is limited by the performance of the underlying networking technology [1]. Consequently, we designed our DPP-E experiments to measure the effect on its performance produced by the selection of the networking technology and by the level of background traffic crossing the network. Moreover, the same literature predicts that it is very likely that the combined power of a NOW based DPP-E may exceed that of a costly supercomputer. Thus, we use experimentally observed performance-measures from a SGI Origin2000 supercomputer as our reference point.

The rest of the paper is organized as follows. In section 2, we describe our DPP-E comprised of the VPM and the transport network. Then, in section 3, we explain our design for the tests and measurements carried out with the DPP-E. Section 4 contains the discussion of observed results. Finally, we conclude and summarize our work in section 5, and section 6 has the bibliography.

2 Distributed parallel processing platform

In this section, we will describe the instrumentation of both our VPM and the underlying transport network. In addition, we will describe the use of a HP Broadband Test system for producing controlled background traffic through the network.

2.1 Virtual parallel machine

We instrumented the virtual parallel machine with PVM (Parallel Virtual Machine) [5]. PVM offers two communication modes, RouteDefault and RouteDirect. In RouteDefault mode, distributed software modules that make up the application interchange messages by means of its local daemon process. In RouteDirect mode, modules communicate directly with each other.

Our VPM has four computing nodes. All are Sun workstations running the Solaris v2.5.1 operating system. More specifically, two workstations are Ultra-1 with 128 Mbytes of RAM, another is a Ultra-1 with 64 Mbytes of RAM, and one more is a SparcStation20 with 64 Mbytes of RAM, also.

2.2 Network configuration

SABA project's ATM network was used as the transport network for the experiment. As any other ATM network [6], a virtual private network (VPN) designed to meet the requirements of a particular service is built upon a physical transport fabric designed to meet topological and logistic requirements.

We designed our VPN to meet the requirements for building a PVM based VPM; that is, IP connectivity. IP connectivity means that the network configuration has to support the delivery of IP datagrams. This implies support for IP to physical-network address resolution and subnetwork routing. Basically, ATM networks may support IP connectivity in two ways: dynamically setting up switched virtual circuits (SVC) through a signaling protocol, such as, LANE [6]; or statically setting up permanent virtual circuits (PVC) that require no signaling protocol. Because using signaling

protocols increase the number of control variables in the experiments, we avoided using SVC's. While using PVC's, we still had another design decision to take. To use a mesh configuration with $2N(N-1)$ PVC's, where N is the number of computing nodes and 2 PVC's make up a full-duplex channel; or to use a star configuration with $2(N-1)$ PVC's. We decided to use a star configuration.

Our local part of SABA's ATM network is composed of two switching nodes and five full-duplex optic channels. Each optic channel runs through a pair of multimode optic fibers, and uses SDH OC-3 physical framing. This means that in each direction optic channels have a 155.52 Mbits per second bit rate, of which 149.76 Mbits per second are available for user data. Fiber optic runs between switching nodes through approximately 50 meters. Then, fiber optic drop cables, between a switch and a workstation, are 3 meters long. This means that the worst case ATM-cell propagation delay is around 0.44 s. ATM switching nodes are realized by one FORE RunnerLE 155 and one FORE ASX-200BX. Each switch is capable of transferring 2.5 Gbits per second [7]. ATM host adapters are from FORE 200 series. Specifically, Ultra workstations use SBA-200E adapters and the SPARCstation uses a PBA-200E. All these adapters are capable to achieve 32-bit Sbus transfers and its embedded logic implements ATM protocols and functions up to AAL5 SAR sublayer [8]. Software support for these host adapters is realized by FORE Throughout tools v4.3.

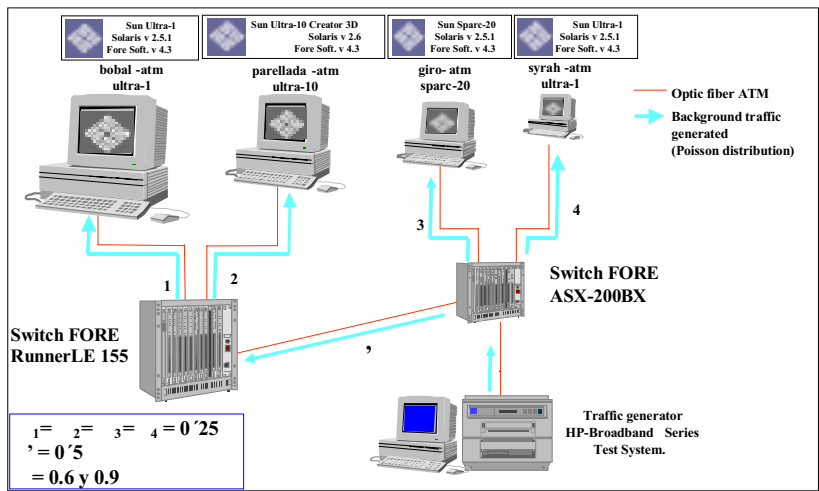


Fig. 1. Background traffic conditions

2.3 Background traffic conditions

In order to assess the effect on the performance of the VPM produced by the selection of the networking technology and by the level of background traffic, we introduced controlled background traffic to the ATM network. Background traffic generation was accomplished by means of a HP Broadband Series Test System (HPBTS). This test system is capable of producing a preprogrammed SDH OC-3 flow of cells, which obeys one of several cell rate and inter-cell time distributions. In order to distribute

the effect produced by the background traffic between the four workstations, we configure the switching nodes as Figure 1 shows. Four simplex PVC's with background traffic flow from the HPBTS to one of each workstation through the ASX switch. This switch forwards two PVC's to the RunnerLE switch and one PVC to each of the two workstations directly attached to it. The RunnerLE switch forwards the received PVC's one to each of the two workstations directly attached to it.

3 Tests and measurements

In order to assess the performance characteristics of the VPM deployed over the SABA ATM network, we designed a set of benchmarking tests. These tests, which we will describe shortly, were thought for comparing our DPP-E against a costly supercomputer. In addition, cause the performance of the VPM is limited by the performance of the underlying telecommunications technology, the tests was designed for assessing this. That is, how does the communication protocol selection affect system performance? Moreover, does the communication protocols take the most of the performance features of the underlying transport network?

A test comprises the execution of a workload hosted by a particular configuration of our DPP-E. We used six configurations grouped in three pairs. Each pair corresponds to a generated background-traffic level, ρ (as shown in Figure 1 and described in 2.3), of zero, 0.6 and 0.9. For each level of ρ , which comprises two tests, one test is carried out with the VPM configured to use RouteDefault (as described in 2.1) and the other configured to use RouteDirect. For comparing purposes, we executed three more tests. One is for defining a reference measure involving the execution of the workload hosted by a SGI Origin 2000 supercomputer. The other two embraces the execution of the workload by a four-node VPM collapsed within one host computer and configured to use RouteDefault and RouteDirect. It is important to note that although the Origin 2000 we used has 64 MIPS R10000 microprocessors, the workload ran there only used four.

Measurements obtained denote registered computing time spent by either the supercomputer or the VPM. We normalized measurement values to the computing time spent by the supercomputer for easing the comparative analysis.

3.1 Benchmarking tests

For benchmarking purposes, we use a set of parallel algorithm realizations extracted from the PVM version of the NAS vLU95 benchmark suite [2]. From the NAS suite we used CG, FT, IS, MG, EP, LU, SP and BT. Table 1 illustrates the communication characteristics of each of these algorithms, accordingly to the distribution of their message lengths. A brief description of these parallel algorithms follows.

Table 1. Distribution of message lengths

Parallel kernels	Parameters used	Number of messages	Length of messages (mean)	Length of messages (median)
CG	1400 matrix size	7116	1963	8
FT	Array of 64^3	186	236776	262144
IS	2^{19} keys in 2^{19} range	599	71364.320	130016
MG	64^3 grid	848	30226.525	2592
EP	2^{23} size	N/A	N/A	N/A
LU	Size 12x12x12	21057	153.006	63
SP	Size 12x12x12	133230	1373	1250
BT	Size 12x12x12	107064	1696	1525

4 Discussion of the results

Table 2 summarizes the registered measurements. We graphically analyzed them in order to assess how does the selection of the networking technology affects the performance of our VPM. In addition, we wanted to see if the communication protocols take the most of the performance features of the underlying transport network. Finally, we wanted to see if our VPM could outperform a supercomputer as predicted in the literature, and under what conditions this does or does not happened.

Figure 2 graphically shows how does the performance of our DPP-E compares against a supercomputer, a computer simulated DPP-E (showed as ATM Ideal) reported in the literature [1], and against a centralized parallel processing environment. This figure shows one graph for each of the workloads use for benchmarking. Each graph depicts computing time spent by the supercomputer and each of the configurations of our DPP-E. In order to inspect the effect that background traffic produces on the performance of our DPP-E each graph shows three curves, one for each level ρ of background traffic. We choose to plot computing time to show that although each workload has different computing requirements the performance of our DPP-E follows similar patterns with respect to the background traffic.

From Figure 2 we can also say that the performance of our DPP-E closely resembles the performance of the computer simulated DPP-E. This is obviously a good thing. Furthermore, the performance of our DPP-E is better than the performance of either of the centralized configurations; that is, the non-parallel (or sequential) and the centralized parallel configurations. By passing, we want to point out that the centralized parallel configuration responds better to increased traffic than the sequential configuration. We think this is due to operating system (OS) scheduling. When we have four processes running over the single CPU, OS scheduling actually pipelines the execution of the problem. This can only happened when computation and communications overlap as in EP, MG, LU, SP and BT. However, in CG, FT and IS this pipelining can be achieved due to communication characteristics of the algorithms.

Comparing the performance between the two configurations of our DPP-E, one using RouteDefault and the other using RouteDirect, we can say that RouteDirect is better than RouteDefault. Because in the first configuration we are avoiding one level

of indirection in the communication path between distributed modules. Moreover, RouteDefault is more sensitive to background traffic, although this is only a slight difference. We think this is because with RouteDirect communication is distributed in $N(N-1)$ TCP connections so congestion in one physical link does not affect the whole communication subsystem.

Figure 3 presents a simplified graphic comparative analysis between the supercomputer and the sequential execution of the workload and the DPP-E. Our DPP-E does not outperform in any case the supercomputer. Nevertheless, it is also true that a supercomputer is far more expensive than an ATM based NOW. In addition, this NOW could be use to solve several other problems not related to high performance computing, such as, computer supported collaborative work or videoconferencing.

Finally, Figure 4 shows the effect produce by background traffic on the performance of the best DPP-E configuration; that is, RouteDirect. There we can see that although performance decreases directly proportional to the level rho of background traffic, communication characteristics of workloads influences the effect produced.

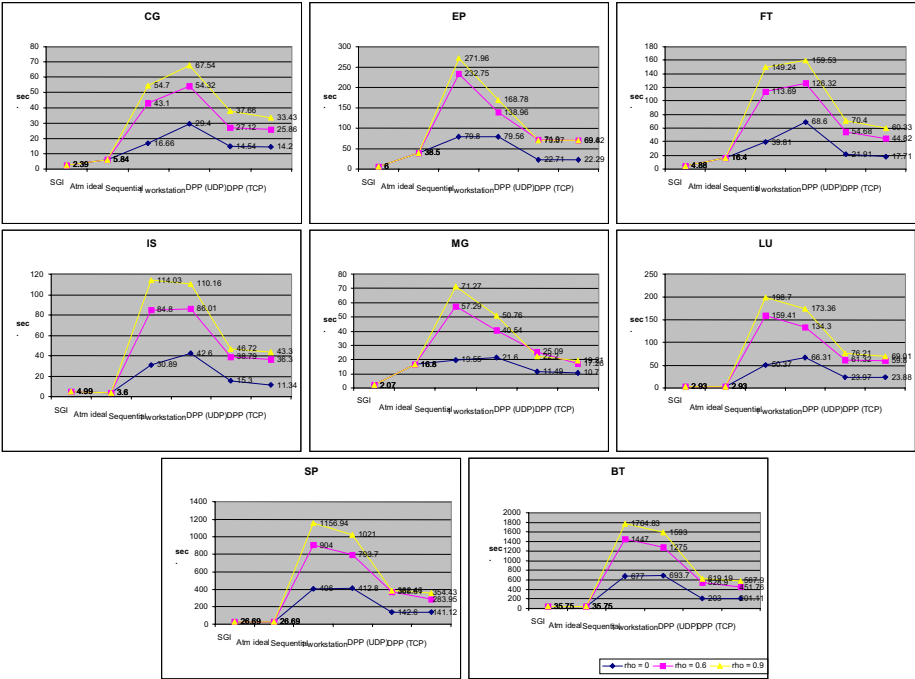


Fig. 2. Computing times for benchmarks hosted by various environments

Table 2. Evolution time of the benchmarks in SGI computer, simulated ATM, and ATM platform with different levels of background traffic.

rho = 0							
	SGI	ideal ATM	1 workstation		4 workstations		
			Sequential	Parallel			
				Default	Direct	Default	Direct
CG	2.39	5.84	16.66	29.4	29.07	14.54	14.2
EP	6	38.5	79.8	79.56	79.95	22.71	22.29
FT	4.88	16.4	39.81	68.6	64.35	21.91	17.71
IS	4.99	3.6	30.89	42.6	39.35	15.3	11.34
MG	2.07	16.8	19.55	21.6	20.73	11.49	10.7
LU	2.93		50.37	66.31	66.66	23.97	23.88
SP	26.69		406	412.8	411.16	142.6	141.12
BT	35.75		677	693.7	690.3	203	201.11
rho = 0.6							
CG			43.1	54.32	53.23	27.12	25.86
EP			232.75	138.96	138.34	71.9	69.42
FT			113.69	126.32	118.19	54.68	44.82
IS			84.8	86.01	79.88	38.79	36.3
MG			57.29	40.54	38.84	25.09	17.26
LU			159.41	134.3	134.15	61.32	59.8
SP			904	793.7	792.61	366.61	283.95
BT			1447	1275	1271.9	528.9	451.76
rho = 0.9							
CG			54.7	67.54	63.38	37.66	33.43
EP			271.96	168.78	167.58	70.07	69.8
FT			149.24	159.53	148.68	70.4	60.33
IS			114.03	110.16	102.1	46.72	43.3
MG			71.27	50.76	48.17	22.2	19.21
LU			198.7	173.36	163.9	76.21	69.01
SP			1156.94	1021	1014	382.46	354.43
BT			1764.83	1593	1483	619.19	567.9

5 Summary

Current ATM-based networks are potentially capable of satisfactory supporting distributed parallel computing applications. The use of such a complex networking technology makes sense when the support parallel computing applications has to be integrated with other services over a single network. Thus, it is not necessary to adopt a specific network for distributed computing, but rather organizations can take advantage of an existing ATM network to support parallel computing in addition to other networking applications. For this reason, a good network-interface design is vital for the network to provide adequate performance to parallel computing. A good design is one that optimizes both protocol-processing latency and congestion-recovery procedures effectiveness.

The experiments show that latency reductions in the network interface can be sufficient to achieve significant performance improvements, provided that the network load from other applications is sufficiently low. In case of extreme congestion situations, loss recovery mechanisms rapidly degrade performance and, consequently, optimizations are required to cover these circumstances. Although currently hard congestion does not seem to be a very frequent issue in ATM networks, the growing trend of integrating multimedia applications over high-speed networks could lead to significant increases of traffic in the networks. Therefore, the need of efficient loss recovery can become evident in the immediate future.

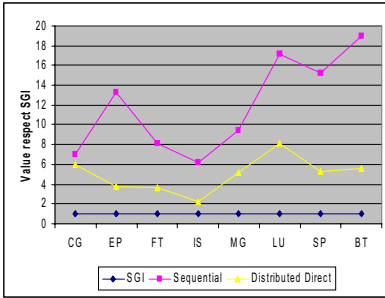


Fig. 3. Comparative analysis between SGI and DPP

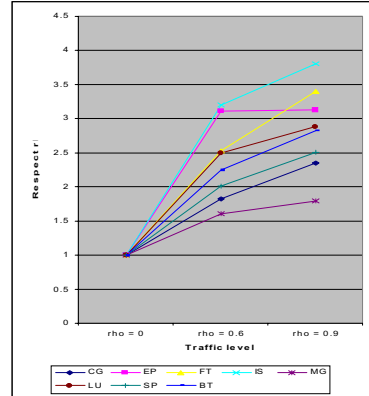


Fig. 4. Traffic effect in the DPP performance

6 Acknowledgments

This work has been supported by CICYT (Spanish Education Ministry) under contract TEL97-1054-C03-03 and by the Mexican Government through CONACyT 66864 grant. The authors want to acknowledge the project SABA for his interest in making this experience possible.

7 Bibliography

- Joan Vila-Sallent and Josep Solé-Pareta, Potential Capability of ATM to Support Network-Based Parallel Computing, Computer Communications Globecom 97.
- S. White, A. Alund and V.S. Sunderam, Performance of the NAS Parallel Benchmarks on PVM Based Networks, Journal of Parallel and Distributed Computing, 26, 1994,61-71.
- Sheue-Ling Chang, David H. C. Du et al., Enhanced PVM Communications over a High-Speed Local Area Network, Distributed Multimedia Center & Computer Science Department, Minnesota, 1995.
- Mengjou Lin, Jenwei Hsieh, David H.C. Du et al., Distributed Network Computing over Local ATM Networks, IEEE Journal on Communication Special Issue of ATM LAN's,13(4),1995,54-64.
- A.Geist et al. PVM 3 User's Guide and Reference Manual, Oak Ridge National Laboratory, 1994.
- Anthony Alles, ATM internetworking, Engineering InterOp, Las Vegas, March 1995.
- FORE Systems, Inc. —Product Catalog, <http://www.fore.com/products/>
- FORE Systems, Inc. Fore Runner SBA-200 ATM Sbus Adapter User's Manual, 1993.
- P.W. Dowd et al, "Issues in ATM support of high performance geographically distributed computing", proceedings of IEEE workshop on high speed computing ,HiNet'95, pp19-28.
- H.Zou et al, "Faster message passing in PVM", proceedings of IEEE workshop on high speed computing ,HiNet'95, pp 67-73.
- P.Papadopoulos et al, "Wide-area ATM networking for Large-scale MPPs", Proceedings of the 9th SIAM conference on parallel processing, March 1997.

Parallelizing of Sequential Annotated Programs in PVM Environment ^{*}

Alexander Godlevsky, Martin Gažák, and Ladislav Hluchý

Institute of Informatics, Slovak Academy of Sciences
Dúbravská cesta 9, 842 37 Bratislava, Slovakia
godlevsky.ui@savba.sk, hluchy.ui@savba.sk

Abstract. In this paper an approach to dynamic parallelizing of coarse and medium grained program is proposed where the parallelization sources are both dataflow analysis and the features given in the program by annotating some of their operators. Program annotation enables to support two additional types of parallel computations which cannot be found out only from the analysis of dataflow dependencies. First, there are the speculative computations based on anticipating alternative branches of the program's computational process. Second, there are pipeline computations that sometimes may be initialized for operators at the moment when their input data are not complete.

The system for dynamic parallelizing of programs is implemented in C++/PVM for PC clusters, and experiments from Buchberger's algorithm are presented.

1 Introduction

Our dynamic parallelizing algorithm is oriented on the class of irregular programs for which it is very hard to schedule computations during the program compilation. Its distinguished features are (a) sources of parallel tasks generated from an executing program, and (b) task control, which results from step-by-step concurrency among some of the tasks. The sources are not only dataflow analysis but also annotations added to some operators of parallelized program. The annotations allow the algorithm to generate tasks executed both speculatively and by the pipeline manner.

The above paragraph allows us to emphasize that messaging in our parallelizing algorithm should be very complex and intensive. Data volume transferred by some messages can be also large, so requirements for the communication platform of our algorithm are very important for the speedup of parallelized program execution. The PVM environment which was created as an application to make up a number of tasks that cooperate to jointly provide a solution of a single problem was chosen as a quite adequate platform to implement our algorithm for a PC cluster.

^{*} This work was supported by the Slovak Scientific Grant Agency within Research Project No.2/4102/98

Irregular programs are usual for such programming paradigms as logic and functional ones, where ideas of speculative [5] and pipeline [2] computations to parallelize programs is developed. In short, the idea of former computations is to compute simultaneously both a function and some of its arguments in the manner when only the function body computation is mandatory. This process determines what to do with each of other processes - to delete it or to make it mandatory. When this process is deleted the computation's overhead is increased; when it is made mandatory computation is speeded up. For function composition $F(..., G(...), ...)$, the idea of pipeline computations is realized by a stepwise execution of both functions F and G . For each execution step only the partial result of G computation is passed to F function to process.

In this paper, combined use of both dataflow analysis and the features of the speculative and pipeline computations is proposed. Programs that will be parallelized are supposed to be written in a simple programming language: each program are composed from unstructured operators (non parallelized procedures) by only sequential composition, *if* and *while* constructions. The characteristic feature of this language is the possibility to annotate some operators of parallelized programs by *spec* and *pipeln* marks. Our system for dynamical parallelization of annotated sequential programs is based on PD semantic proposed in [4]. It guarantees that results of sequential and parallel computations of the same program are equal to each other. The idea of such program parallelization was to reduce the process of parallel programs development to the process of annotating sequential ones.

The main result of this paper is to describe the PVM implementation of dynamical parallelization and experiments on it.

2 Parallel Dynamical Semantic

In this section the sketch of *PDS* semantic is given. It is represented by finite system S of transition rules which act on the states - abstractions of the available states of programs during their parallel execution. A state of S system is 4-tuple $(b, heap, pheap, P)$ where b is a current state of the program environment, P is a residual program, *heap* and *pheap* are sets of unstructured operators extracted earlier from P each subset of them can be executed in parallel. The b state is interpreted as functional $X \rightarrow D$ where X is the set of all variables of parallelized program and D is their data domain. For initial S states, *heap* and *pheap* components are empty sets, P is parallelized program and b state corresponds to its input data. All components of a final S state, are empty sets, with except b that represents output data. There are four groups of transition rules: for transferring unstructured operators from P program into *heap* or *pheap* components, for reducing P according to semantic of *if* and *while* constructions, for execution unstructured operators both in pipeline and usual modes, for eliminating speculative operators or transforming them in mandatory ones. The transition rules are conditional, and take into consideration data flow dependences of anal-

ized operators, presence of *spec* or *pipeln* marks and values of some intermediate predicates which is not viewed on users level.

Operator y is transferred from P into *heap* or *pheap* if it is dataflow independent to all operators which are in *heap* or *pheap* sets or precede y in P . *heap* and *pheap* differ by the feature that operators from *pheap* can be executed in the pipeline mode. The notion of dataflow independence is modified for them: operators R and Q , where R precedes Q and Q is annotated by mark *pipeln*(x), are independent if the set intersection of output variables of R and input ones of Q contains the only variable x whose value is step by step transferred from R into Q . When the execution rule is applied to unstructured operator y from *heap*, y is deleted from *heap* and b is transformed according to the semantical interpretation of y . The similar rule for pipelined y operator from *pheap* differs in that y may be deleted from *pheap* only then the computation of its input variable executed step-by-step is finished.

A speculatively executed y operator differs from that with mandatory execution by executing in b environment extended by local variables. If y 's speculation, for some y from *heap*, is found to be useless, then its environment extension is deleted. If it is turned to be mandatory then the b environment is changed by decreasing localisation level of its local variables. Being limited by the paper volume we cannot demonstrate accurately any of the transition rule. We only note that the correctness of their system was proved in [4]. The order of rules' application is limited only by conditions of their applicability. The potential differences in the order can affect computation speed-up but not correctness.

3 Implementation

The system for dynamic parallelization of programs (*SDP*) is implemented in C++/PVM. PVM provides basic communication and synchronization services, on which simplified Remote Procedure Call services implementation is based. Since PVM became de facto standard in network and distributed computing, there are many tools for monitoring and visualization of application performance developed. However, *SDP* is suitable for other implementation languages (Java, etc.).

Current version assumes a distributed memory homogenous message passing machine (identical nodes and the same communication costs for any pair of nodes). The target machine for *SDP* implementation is a cluster of PCs running Linux using fast ethernet hub.

Parallel program execution consists of two steps:

1. Compilation: the source program consists of skeleton of algorithm written in a modified subset of C language enriched with keywords for *pipeln* and *spec* annotations and of implementation of unstructured operators. During the compilation phase two pairs of C++ files are generated : *resolver.cxx*, *resolver.h* implements remote procedure call via PVM, and *dpc_program.cxx*, *dpc_program.h* initializes representation of program in a form of linked lists of statements (unstructured operators, while, etc.).

2. Interpretation and parallel execution: the input for dynamic paralleliser consists of object of the *Program* class and of the initial memory state. Program is interpreted according to *PDS* rules presented in section 2.

SDP has a star topology (Fig. 1). It consists of:

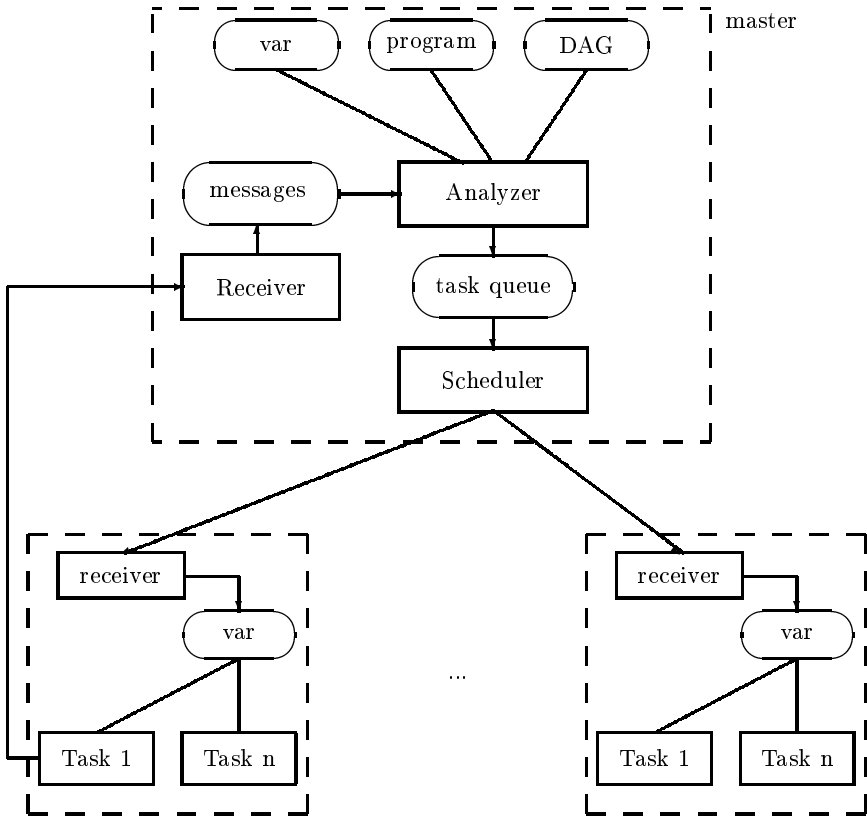
1. Master process running on the main processor and consisting of two threads - the *analyzer* and the *scheduler*; within this process the interpretation and parallelization of sequential program take place. Analyzer and scheduler run concurrently and share data structures. An access to those data structures is synchronized by synchronization mechanisms - monitors and synchronized methods.
2. Slave processes on the slave processors. Slave processes communicate with the scheduler, manage the task threads (a *task* is an unstructured operator under execution) and the local memory of variables shared among the tasks (an example of shared variable is a list used for pipeline parallelism).

During the interpretation the analyzer analyzes dependencies among operators and gradually transforms program into directed acyclic graph (*DAG*), where nodes correspond to unstructured operators and edges correspond to dataflow dependencies (data output dependencies and antidependencies are broken during the analysis according to *PDS* rules). As soon as for any node the number of incoming edges is 0 (or 1 for nodes representing pipeline operators), the operator is moved to corresponding task queue (priority of mandatory operators is higher than that of speculative ones, etc.). *DAG* is generated from sequential program in run-time, programs with conditional statements (*if-then-else*, *while*) cannot be transformed into *DAG* during one pass. The analysis starts at the beginning of program, then links between operators from *DAG* and conditional statements, which depend on the values of output variables, are established, so that after the execution of an operator the analysis need not start at the beginning. The analyzer also manages the memory and takes care of variable localization and garbage collecting of unnecessary copies of variables.

The scheduler manages the whole system. During the initialization it establishes connections to slave processes, launches an analyzer and then manages the tasks. The tasks wait for execution in several queues with different priority. The scheduler does not have information about task granularity, and simple FIFO scheduling algorithm is used. The scheduler maps the task on the first free processor (with respect to load). Since values of output variables of operators are necessary for further analysis, and tasks which became useless (e.g. during speculative parallelization) are destroyed, all communication is between task and scheduler, not between the tasks themselves.

4 Buchberger's Algorithm

The Buchberger's algorithm for the computation of Gröbner bases is one of the fundamental algorithms to solve polynomial system. From the perspective of


 Fig. 1. Processes of *SDP*

computational complexity the algorithm is intractable, but in practice it can solve a considerable number of interesting problems and there are indications that problems arising from real situations are far from the worst case of algorithm.

Let K be a field, and assume we have a term-ordering in polynomial ring $K[x_1, \dots, x_n]$; a power-product in this ring is the product of variables, a monomial is a product of non-zero constant from K and a power-product. The leading power product $Lpp(p)$, the leading monomial $Lm(p)$ and the leading coefficient $Lc(p)$ of polynomial p are defined w.r.t. the term-ordering. We say that p reduces to p' by q at τ , and write $p \rightarrow_q^\tau p'$ if $p = a\tau + \rho$ and $q = bv + \eta$, with $a, b \in k$ constants, τ and v power-products such that $v = Lpp(q)$ and $v\mu = \tau$, then $p' = b\rho - a\mu\eta$.

The polynomial p had reduced relatively to the polynomial set P if any its polynomial does not reduce it. In this case it is also said that polynomial p has the irreducible form w.r.t. to P . Let P be a polynomial set, $Id(P)$ be the ideal generated by P over the polynomial ring $K[x_1, \dots, x_n]$; then a polynomial set G such that $Id(G) = Id(P)$ and irreducible form of each polynomial from P is 0 will be named Gröbner's basis of P . Buchberger's algorithm for construction of Gröbner's basis is based on the notion of S -polynomial. In [3] it was proved the basis G is Gröbner's basis if and only if for each pair of polynomials from G its S -polynomial is reduced to 0 relatively to G .

Gröbner's basis is not unique but if the reduction of each basic polynomial will be done in such a way to each monomial of each basic polynomial would not be divisible by any leading monomial of another basic polynomial; then such Gröbner's basis will be unique and will be called reduced.

```

Program Groebner (inout  $S$ :set of poly);
Pairs := {(f,g)|f,g ∈ S, f < g};
while not isempty(Pairs) do
  begin;
    Var f,g,p:poly;
    (f,g) := Select(from Pairs);
    Pairs := Pairs \ {f,g};
    p := Spol(f,g);
    p := Red(p,S);
    if not eq(p,0) then
      S := S ∪ {p};
      Update({p} * S to Pairs);
    fi;
  end_block;
end_while;
S := Reduce(S);
end program

```

Fig. 2. Buchberger's algorithm

It will consider the Buchberger's algorithm version given on Fig. 2 which has only unimportant differences to one from [1]. These differences consist in the programming language we used and the presence of the *Reduce* procedure to transform basis to the reduced form. The algorithm contains the calls of operators (procedures) *Select*, *Red*, *Reduce*, *Update* and *Spol* which have the following meaning. The *Select* selects an element from *Pairs* set. The *Red* reduces polynomial p relatively to the current value of the S basis. The *Update* completes the *Pairs* set by new elements applying the B-criterium. The *Spol* constructs S -polynomial for its two arguments. *Spol*, *Red*, *Reduce* are marked as unstructured operators, i.e. only these operators will be computed in other-scomponents of multicomputer concurrently with interpreting of the algorithm itself.

Because of data dependencies among the loop iterations this algorithm cannot be parallelized in the framework of the conservative approach. Annotating this loop by *spec(Pairs)* mark we open the possibilities to initiate speculative parallel computing of certain iterations. If for some element of *Pairs* speculative computing is initiated and after this moment according to B-criterion [1] for current completing of *Pairs* this element is eliminated, then speculative process initiated by it will also be killed by corresponding mechanism of *SDP*.

Buchberger's algorithm is nondetermined in three ways: the selection order from *Pairs*, the selection order of polynomial from *S* set during reducing of current *p* polynomial, and the selection order of interreductions can vary. It is evident that the following relationships for *Red* and *Reduce* operators,

$$Red(p, M \cup N) \cong Red(Red(p, M), M \cup N)$$

$$Reduce(M \cup N) \cong Reduce(Reduce(M) \cup N)$$

where the sign \cong is understood as equivalence, are true. Thus, these operators are pipeline-computing.

5 Experimental Results

In this section the results from Buchberger's algorithm running on PC cluster are presented. During the parallel execution one processor interpreted the program and the rest of processors computed unstructured operators. Time for one processor is a time of execution of sequential version (without overhead caused by interpretation). Times are in seconds. Benchmark Quad consisted of 11 polynomials of 7 variables (degree 2, 435 pairs of polynomials), Symm 2 of 3 symmetric polynomials of 4 variables (degree ≤ 3 , 36 pairs), Nonsymm of 3 polynomials of 3 variables (degree ≤ 3 , 120 pairs).

Number of processors	Quad(time /speedup)	Symm 2	Nonsymm
1	44.8 / 1	8.04/1	223/1
4	16.47 / 2.72	3.1/ 2.59	78.4 / 2.84
8	7.1 / 6.31	1.82 / 4.45	32.84 / 6.79

Configuration of slave processors permitted computation of several tasks (in separate threads) on one processor concurrently. With increasing number of threads per processor the behaviour of simulation became more nondeterministic because of communication collisions; in general the speedup was the same. Granularity of tasks of Quad example is the lowest, the one of Nonsymm example is the highest. Symm2 shows how efficiency is affected not only by granularity of tasks, but also by their number as well. The analysis of system performance is based on the data obtained from PGPVM (see [7], monitoring tool for PVM).

6 Conclusion

PVM based implementation of system for automatical parallelization of annotated sequential programs was represented. Its dynamical manner of processing gives opportunity to adapt the parallelization process not only to parallelized programs but to their input data. As a consequence, the system can take its place for parallelization of irregular programs to what is not applied the developed methods of data parallelization. We think that the restrictions of parallelization process followed from using of master-slave model will not be valuable for coarse and medium grained irregular programs. The results of experiments with PVM implementation on PC cluster showed that efficiency ((number of processors * parallel time)/sequential time) achieved 50 % - 85 %. This demonstrates a suitability of PVM as a communication platform for such projects.

References

1. Attardi G., Traverso C.: *A Strategy-accurate Parallel Buchberger Algorithm*, J. Symbolic Computation (1996), 21, P.411-426
2. Guy E.Blelloch, Margaret Reid-Miller, *Pipelining with Futures*, in Ninth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'97), 22-25 June, Newport, Rhode Island
3. Buchberger B.: *An Algorithm for Finding a Basis for the Residue Class Ring of Zero-Dimensional Polynomial Ideal*. Ph.D.Thesis, Math.Inst., Univ. of Insbruck, Austria, 1965.
4. Godlevsky A.B.: *The Parallel Dynamical Semantics of Sequential Program that Allows Speculative and Incremental Computation*. Kibernetika i sistemny analiz , 1996, No. 2, pp.131-153 (in Russian).
5. M.Hermenegildo.: *Automatic Parallelization of Irregular and Pointer-Based Computation: Perspectives from Logic and Constraint Programming*, in Euro-Par'97 Parallel Processing, number 1300 in LNCS, pages 31-45, Passau, Germany, August 26-29, 1997. Springer-Verlag
6. A. Geist, A Beguelin, J.Dongarra, W.Jiang, R.Manchek, V.Sunderam: *PVM : Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, 1994
7. Brad Topol, V. Sunderam and Anders Alund: *PGPVM performance visualization support for PVM*, Technical report CSTR-940801, Emory University, August 1994

Parallel NLP Strategies Using PVM on Heterogeneous Distributed Environments

Gustavo E. Vazquez¹ and Nélida B. Brignole²

¹ Dept. of Computer Science, Universidad Nacional del Sur, Bahía Blanca, Argentina
Phone: 54 0291 4861700, Fax: 54 0291 4861700, e-mail: gvazquez@criba.edu.ar

² Planta Piloto de Ingeniería Química, UNS-CONICET, Bahía Blanca, Argentina
Phone: 54 0291 4861700, Fax: 54 0291 4861700, e-mail: dybrigno@criba.edu.ar

Abstract. In this work we present a strategy for the parallelization of gradient-based nonlinear programming techniques, which is particularly suitable to solve optimization problems whose objective functions and/or constraints are costly to evaluate. The algorithm was conceived to be run on heterogeneous distributed computing environments. In order to avoid communication overheads and improve load balancing, we propose the use of a hybrid computational task-scheduling model.

1 Introduction

Parallel processing is a well known technique that enables significant reductions in execution time. Most of the literature about the topic concentrates on the design of algorithms for Massively Parallel Computers (MPPs). Nevertheless, the applicability of this approach is limited by the need for expensive equipment and the lack of manufacturer standards. In contrast, a distributed configuration of workstations connected by local communication networks allows the efficient use of existing resources. This parallel configuration offers minimal start-up budget and easier scalability, not only to boost the speed of computation but also to accommodate larger problems in a distributed memory environment.

This paper is focused on the application of parallel optimization on distributed computing environments to constrained nonlinear problems that commonly arise in chemical engineering. One of the main characteristics of most of these problems is that the evaluation of the objective function and/or the constraints is computationally expensive. In most cases, this stage becomes costly because each evaluation implies simulating the whole process. This feature makes these problems particularly apt to be solved under distributed environments. In this work, we present a parallel implementation based on the Successive Quadratic Programming (SQP) optimization technique. For message transfer, we have employed the Parallel Virtual Machine library (PVM) [1]. Though focused on a specific optimization method, many aspects of both the parallelization and performance evaluation strategies discussed in this paper can be generalized in a straightforward way and, therefore, may prove to be useful for the future development of other parallel-distributed optimization codes. This is a promising research field which is emerging nowadays and has never been applied to the optimization of realistic chemical engineering problems.

2 Parallel Optimization Background

As regards parallel optimization, there are two different approaches:

- (a) Intrinsic parallel optimization algorithms.
- (b) Sequential optimization algorithms, later transformed into parallel ones.

In the first case, the techniques exploit parallelism by solving the algorithm on distinct data items at the same time. Each processor has a complete copy of the algorithm and the problem domain is divided amongst the processors. Several authors ([2], [3], [4]) followed this methodology, basically addressing the same approach, which consists in splitting the optimization variables into p blocks and defining p associated subproblems. Each of them is solved in parallel and these intermediate results are used to produce a better solution; this procedure is repeated until convergence. These works only considered the unconstrained case and problems with block-separable constraints.

In the second approach, the parallel optimization algorithm is developed by introducing parallelism at various points of a sequential algorithm for the same purpose. High and LaRoche [5] suggested an SQP algorithm, where they parallelized both the line search procedure and the derivative calculations via simultaneous function evaluations at different processors. A reduced Hessian SQP algorithm was proposed by Ghattas and Orozco [6] for aeronautic design applications. All these proposals were implemented on parallel machines.

The optimization methods based on domain decompositions are not suitable for chemical engineering problems because they cannot deal with nonlinear models, no matter if they have been transformed by means of modifiers such as penalty functions. At the same time, the parallel algorithms based on sequential techniques inherit desirable features concerning efficiency and robustness. No author has addressed, however, the application of these methodologies to parallel heterogeneous environments.

3 Parallel Optimization on Distributed Computing Environments

Let us consider the general nonlinear programming problem:

$$\begin{aligned} & \min f(x) \\ & \text{s.t. } g_i(x) \geq 0 \quad i = 1, \dots, ic \\ & \quad h_j(x) = 0 \quad j = 1, \dots, ec \end{aligned}$$

where g and h are inequality and equality constraints, ic and ec are the respective numbers of constraints.

As we mentioned above, the high computational cost involved in the evaluation of the model functions constitutes a distinctive feature of optimization problems in the field of chemical engineering. The completion of the evaluation tasks typically requires much more time than the rest of the calculations, which are relatively fast. This is so because each evaluation most often requires the

numerical simulation of the entire process plant at a given set of input conditions, which changes at every major iteration. This may imply the need to solve many thousands of equations. As a result, the sections of the algorithm that carry out repetitive evaluations become the hot spots, dominating the total execution time. The procedures for gradient and Hessian evaluation, together with the line-search stage, constitute unavoidable hot spots that arise in all the main nonlinear optimization strategies.

3.1 General Algorithm

This is a global scheme for all the methods that generate search directions:

1. $k = 0$
2. Check for convergence
3. Calculate the direction vector d_k
4. Find α_k so that $f(x_k + \alpha_k d_k)$ is a minimum, $x_{k+1} = x_k + \alpha_k d_k$
5. $k = k + 1$; go to 2

The use of an SQP technique, in particular, would benefit from the incorporation of parallelization at the following stages of this algorithm:

1. Step 2 is typically implemented by checking whether the norm of the gradient of the Lagrangian function $\nabla_x L(x, u, v) = \nabla_x (f(x) - \sum_{i=1}^{ec} v_i h_i(x) - \sum_{i=1}^{ic} u_i g_i(x))$ as well as the norm of the active-constraint vector is less than a user-specified tolerance value. Here u and v are the Lagrange multipliers. The active constraints comprise the equality constraints and those inequality constraints that are either violated or equal to zero (binding).
2. Step 3 involves the solution of a problem with quadratic objective function and linear constraints, which can be formulated as follows for iteration k :

$$\begin{aligned} & \min f(x^{(k)})^T d + \frac{1}{2} d^T H^{(k)} d \\ & \text{s.t.} \\ & g_i(x^{(k)}) + \nabla g_i(x^{(k)})^T d \geq 0 \quad i = 1, \dots, ic \\ & h_j(x^{(k)}) + \nabla h_j(x^{(k)})^T d = 0 \quad j = 1, \dots, ec \end{aligned}$$

where $H^{(k)}$ is the chosen approximation for the Hessian matrix of the Lagrangian function L . Powell [7] suggests updating the Hessian by means of the BFS formula, which ensures that $H^{(k)}$ always remains positive definite:

$$H^{(k+1)} = H^{(k)} - \frac{H^{(k+1)} z z^T H^{(k)}}{z^T H^{(k)} z} + \frac{w w^T}{z^T w}, \quad H^{(0)} = I, \quad \text{where} \quad (1)$$

$$y = \nabla_x L(x^{k+1}, u^{k+1}, v^{k+1}) - \nabla_x L(x^k, u^{k+1}, v^{k+1}) \quad (2)$$

$$z = x^{k+1} - x^k, \quad w = \theta y + (1 - \theta) H^{(k)} z \quad (3)$$

The QP problem cannot be parallelized in a straightforward way and this is not worthwhile because the time savings would be negligible in comparison

with the gains achieved through Steps 2 and 4. Nevertheless, it is advisable to parallelize the gradient calculations in the updating formula.

3. In Step 4, a common strategy is to minimize the penalty function $P(x, \mu, \sigma) = f(x) + \sum_{i=1}^{ec} \mu_i |h_i(x)| - \sum_{j=1}^{ic} \sigma_j \min(0, g_j(x))$. This calculation is expensive because it involves the determination of values for the objective function and the constraints. So, significant time savings can be obtained at this stage by parallelization of the line-search procedure.

3.2 Parallel Numerical Gradient Evaluation

The gradient-based optimization techniques employed information about the first derivative of the functions involved. In many realistic situations, the functions cannot be written in an analytical way. Therefore, the derivatives are to be calculated by numerical differentiation. There are several formulas to approximate the first derivative. In this work we employed the forward-difference approximation, which emerges directly from the limit definition of a derivative:

$$\frac{\partial f(x)}{\partial x_i} = \frac{f(x + \delta_i e_i) - f(x)}{\delta_i} + O(\delta_i), i = 1, n \quad (4)$$

where δ_i is presumed small, e_i is the i th versor, n is the dimension of vector x , and the last term refers to the magnitude of the approximation error, which is proportional to δ_i . All the function evaluations involved in these calculations can be carried out efficiently by using parallel processing. This formulation requires the evaluation of only one function at each $(x + \delta_i e_i)$ as well as at x . Thus, for an n -dimensional problem $n + 1$ tasks are necessary in order to calculate a gradient. Function evaluations within a derivative calculation are totally independent from each other. Therefore, it is possible to estimate them simultaneously.

3.3 Parallel Line-Search

At the initial phase, the optimum point must be bracketed within an interval. This bounding search is conducted by using a heuristic-expanding pattern. An example is Swann's method, where $k + 1$ test points are generated by means of the recursive formulation $x_{k+1} = x_k + 2^k d \delta$, for $k = 0, 1, 2, \dots$, where x_0 is the starting point, d is the direction vector of the line-search and δ a positive step-size parameter of a suitably chosen magnitude. The search is over when $f(x_{k-2}) \geq f(x_{k-1}) \leq f(x_k)$, for $k \geq 2$. A quadratic estimation scheme [7] assumes that in a bracketed interval the function can be approximated by a quadratic function and that this approximation will improve as the points used to build it approach the actual optimum. The bounding step must sometimes be repeated within the approximation region in order to refine it and obtain a better estimate. This stage can be parallelized performing the evaluations in advance.

3.4 Parallel Function Evaluations in a Distributed Environment

Major reductions in optimization time can be achieved by carrying out function evaluations for gradients and line-search calculations simultaneously. A classical way of implementing the corresponding parallel algorithms is the use of a MASTER-WORKER scheme, where the MASTER schedules and collects the function evaluations done by the WORKER processes. A sound load-balancing technique is necessary in order to provide an even division of this computational effort among all processors. Nevertheless, a parallel implementation in a heterogeneous distributed environment must consider the following aspects:

- (a) The computing performance of each workstation may be different.
- (b) All processes carrying out a parallel task must compete with other foreign processes currently running at the workstations, which results in unpredictable levels of performance during run-time.
- (c) The time required to evaluate a function differs at every major iteration

In a data-driven scheduling model, the parallel tasks such as function evaluations are scheduled by the MASTER process to the WORKERS at the beginning. Assuming that all these tasks last the same time, the time effort needed to solve them is predicted (by using, for example, a measurement based on the previous evaluations) and the computer performance of all the processing nodes is known in advance, it is simple to determine a static scheduling of the work. However, as no further allocation takes place after the initial distribution, a balanced work load cannot be guaranteed in situations involving (b).

In the demand-driven computational model, tasks are allocated to WORKERS dynamically, i.e. as they become idle. Having evaluated a function, they demand another task from the work supplier process (MASTER). Unlike the data-driven model, there is no initial communication of tasks to the WORKERS. Instead, there is a need to send requests for individual function evaluations as well as the corresponding result messages informing that a processor is idle.

Although the demand-driven computational model fits better in a dynamic environment such as the one arising in this application, the communication overhead may become prohibitive. The scheduling of a single task (function evaluation at x) comprises a message to ask for the evaluation, the effective time to evaluate the function at x and a message to inform about the function value. If the function-evaluation time is close to the sum of times of the messages, which is the case, for example, on heavily loaded networks, a lot of time is wasted in communication instead of performing useful computation.

For this reason, we propose the use of an hybrid computational model for task scheduling. The main idea is to allow a queue of pending tasks at WORKER processes. This avoids the communication overhead of the demand-driven approach by allowing a continuous supply of work: the WORKER reads the next point to evaluate in its local queue, performs the calculations, and returns the value to the MASTER. Since the last message can be sent in a non-blocking way, the process is able to serve the next requirement immediately. Obviously, we should ensure that its local queue is occupied during the parallel execution.

The considerations (a)-(c) mentioned above imply different local queues lengths for each processor. Moreover, this length limit might be modified at run-time in the presence of (b). This changing behavior of the environment and the application involves the use of a dynamic profile scheme by MASTER process to monitor the parallel running and schedule the tasks. We present the following algorithm for task scheduling using the proposed hybrid computational model:

```
while(no more function evaluations to process)
  Receive a function-evaluation message from processor i
  Update  $PM_i$  and  $CO_i$  metrics
  Update queue length limit  $QLL_j$ ,  $j:1..number\ of\ SLAVE\ processors$ 
  if appropriate,send next (one or various) function evaluation
    message to processor with available queue
```

where PM_i is a performance metrics, QLL_i the queue length limit of function evaluation messages and CO_i the communication time overhead to send and receive a message (to-from) i SLAVE processor. PM_i can be calculated by averaging the m last function-evaluation times for the i SLAVE processor, where m is given. This ensures a better fitting of the metrics to the dynamic behavior. The initial distribution of tasks into WORKERS can be done by using a data-driven approach to obtain the first PM_i .

4 Implementation and Results

In the first place, we implemented a general SQP optimization algorithm in FORTRAN language. It is based on Powell’s strategy [7], which updates the Hessian by means of the well-known BFS formula. The numerical gradients were evaluated using a forward-difference approximation formula and the QP problem was solved by the freely distributed code QL0001 (Version: 1.4).

In order to assess the savings achieved through parallelization, we compared sequential and parallel results for the following four nonlinear optimization problems selected from Hock and Schittkowskis [8] test suite. In this work, the corresponding objective functions were overloaded to simulate the models that arise in chemical engineering areas. Table 1 identifies and characterizes these test problems. As regards the parallel algorithm, we built it by adding the parallelized

Table 1. Test problem specifications.

Test Problem N	Description	Number of Variables	Number of inequality constraints	Number of equality constraints	Number of Bounds
78	Powells problem	5	0	3	0
113	Wongs problem 2	10	8	0	0
114	alkylation problem	10	8	3	20
117	Shell Dual problem	15	5	0	15

procedures described in Section 3 to the frame of the sequential SQP. With respect to the implementation, we had to keep in mind that the code should be

efficient when working on a network of heterogeneous workstations. This means that we should employ a software tool that provides communication facilities and manages heterogeneity in a transparent way. For this reason, we chose the PVM (Parallel Virtual Machine) message-passing interface [1]. It operates on a heterogeneous collection of processing units (workstations, vector or even parallel computers) interconnected by one or more networks.

The PVM features which proved to be necessary for this specific application include: dynamic start-up of slave processes, non-blocking "send" and "receive" operations and test for incoming messages. The parallel version was also written in FORTRAN, using the library provided by the PVM distribution. The runs were carried out using a 400 MHz PC PentiumII (WS 1), a 133 Mhz PC Pentium (WS 2) and a 150 Mhz DEC ALPHA (WS 3) connected by 10Mb Ethernet network. The Intel-based workstations utilize LINUX operative system, whereas the DEC ALPHA uses OSF/1.

Table 2 shows the number of iterations each example required for convergence, followed by the sequential times taken by each workstation. The last two columns contain the parallel time and relative speed-up achieved. WS 1 ran as MASTER process and the three machines executed SLAVE processes. Regarding the speed-up, we devised the weighted speed-up (WSU) measurement to take into account the heterogeneity in computing powers:

$$WSU = \frac{\sum_{i=1}^p ST_i * CPF_i}{TM}, \quad CPF_i = \frac{WCP_i}{\sum_{j=1}^p WCP_j}, \quad WCP_i = \frac{\sum_{j=1}^p ST_j}{ST_i} \quad (5)$$

where p is the number of processors, TM is elapsed time for parallel computing and WST_i , ST_i , CPF_i , WCP_i are the weighted elapsed sequential time, the elapsed sequential time, the computing power factor and the weighted computing power for the i -th processor respectively.

Table 2. Parallel algorithm performance.

Test Problem N ^o	Number of Iterations	Seq. time WS 1	Seq. time WS 2	Seq. time WS 3	Parallel time	Weighted Speed-up
78	5	16.3s	50.3s	21.4s	8.1s	2.89
113	13	59.6s	3:10.1s	1:17.23s	30.2s	2.82
114	6	30.8s	1:36.0s	36.3s	14.8s	2.87
117	20	2:0.2s	6:5.2s	2:36.9s	58.1s	2.81

The weighted speed-ups are satisfactory because the values are close to three, which implies more than 90% relative efficiency. After parallelization, the only sequential steps left were the QP solver and the basic algebraic operations. As regards computing times, these stages are not affected by the complexity of the function evaluation procedure. Therefore, speed-ups are expected to increase as the objective function becomes more costly to evaluate. In turn, the gains for simple objective functions will be low because communication costs become dominant in those cases.

5 Conclusions

We have analyzed a general framework for the development of parallel nonlinear optimization algorithms with costly objective functions, such as those that commonly arise in many chemical engineering applications. In particular, we proposed a hybrid computational model for the scheduling of multiple function evaluations and we implemented it for an SQP optimization algorithm, making use of the facilities provided by the PVM message-passing library. The general philosophy was to employ well-known sequential optimization techniques as a basis for the creation of robust and reliable parallel codes.

A significant contribution of this work is the fact that the algorithmic design and analysis were specially adapted for heterogeneous distributed environments. Besides, the code was tested on a variety of nonlinear optimization problems where the function objective was artificially overloaded to simulate the complex models that arise in realistic applications. We obtained satisfactory results and quantified them by means of a new weighted speed-up formula.

Acknowledgments

We acknowledge the economic support given by ANPCyT and CONICET, Argentina, through grants PICT97 03-00000-01258 and PEI 0068/98 respectively.

References

1. Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R., Sunderam V.: PVM : Parallel Virtual Machine. Users' Guide and Tutorial. MIT Press. (1994)
2. Ferris M.C., Mangasarian O.L.: Parallel Variable Distribution. *SIAM Journal of Optimization* **4** (1994), 815–832
3. Mangasarian O.L.: Parallel Gradient Distribution in Unconstrained Optimization. *SIAM Journal on Control and Optimization* **33(6)** (1995), 1916–1925
4. Mittelman H.: Parallel Multisplitting for Constrained Optimization. *Parallel Algor. Appl.* **9** (1996), 91–99
5. High K.A., LaRoche R.D. Parallel Nonlinear Optimization Techniques for Chemical Process Design Problems *Computers Chem. Engng.* **19** (1995) 807–825
6. Ghattas O., Orozco C.E.: A Parallel Reduced Hessian SQP Method for Shape Optimization Tech. Report, Comp. Mech. Lab, Carnegie-Mellon University (1996).
7. Reklaitis G., Ravindran A., Ragsdell K.: *Engineering Optimization*. J. Wiley (1983)
8. Hock W., Schittkowski K.: *Test Examples for Nonlinear Programming Codes*. Springer-Verlag, New York (1981)

Using PVM for Distributed Logic Minimization in a Network of Computers

Luis Parrilla¹, Julio Ortega², and Antonio Lloris¹

¹ Departamento de Electrónica y Tecnología de Computadores.

² Departamento de Arquitectura y Tecnología de Computadores
Facultad de Ciencias. Universidad de Granada, E-18071-Granada, Spain
{lparrilla, lloris}@ditec.ugr.es, jortega@atc.ugr.es

Abstract. The implementation of switching functions by using the AND and EXOR primitives makes it possible to manufacture circuits which are easier and faster to test. This paper describes a parallel procedure that accelerates the execution of an AND-EXOR logic minimization procedure previously proposed. It is based on the use of Simulated Annealing (SA) and rewrite rules. The parallel procedure presented uses multiple Markov chains with periodic exchange of information to distribute the SA process, on which the sequential AND-EXOR minimization procedure is based.

1 Introduction

In this paper, a parallel two-level AND-EXOR minimization procedure is presented. The design of digital circuits with AND-EXOR gates [1] uses fewer gates than those required by an AND-OR implementation, and testing is faster and easier to generate [2]. The high complexity implied by the AND-EXOR minimization problem (it is an NP-complete problem [3]), implies high computing times in the procedures which are able to obtain satisfactory results. Thus, the use of parallel computers and networks of computers for these procedures would require less time to reach good solutions, allowing the AND-EXOR logic to be considered as a true alternative in the design of a digital circuit without requiring a high amount of time.

The parallel procedure is based on RRMIN2 (Rewrite Rules MINimizer 2) [4], which provides excellent results by using a convergent set of rewrite rules [5] whose application is controlled by a Simulated Annealing (SA) process [6]. Thus, the parallelization of RRMIN2 implies the parallelization of the SA, which is achieved through multiple Markov chain with periodic information exchange.

The parallel minimization procedure, called PRRMIN (Parallel Rewrite Rules MINimizer), has been implemented by using PVM message passing libraries [7] on networks of workstations, which present a very good trade off with respect to cost and performance, and are widely available in circuit-design departments. Thus, it is possible to include this kind of parallel minimization procedure as another option in the available CAD tools.

2 The RRMIN2 Procedure

The RRMIN2 procedure is based on using a SA procedure that controls the application of a set of rewrite rules, as is described below. The SA procedure used has been suited to be applied in this problem by optimizing both the SA parameters and the selection process of the rewrite rules used. In this way, the AND-EXOR minimization problem has been formulated as a combinatorial optimization problem [8], defining a solution space and a cost function [4]. The procedure to solve this optimization problem requires two mechanisms: (a) Generate new solutions from a given one, and (b) Accept or reject the new solution generated, according to the cost function optimization offered by such a solution.

The application of rewrite rules is used to generate new solutions from a given one. In order to avoid restrictions in the set of possible new solutions, a convergent set of rules should be considered [5]. A SA algorithm [8] is used to accept or reject each new generated solution. The selection of a SA algorithm as the acceptance mechanisms is justified by the good results provided when applied to other combinatorial optimization problems [9,10,11]. The results provided by this procedure outperform those reported by other authors as shown in [4]. Nevertheless, the computing time required, although is not very great (about 45 seconds for a logic function of 8 variables), is higher than those of the other procedures. Thus, it would be beneficial to improve the required computing times.

The RRMIN2 flowchart is provided in Figure 1, where the approximate percentages of time taken by each task are also indicated. As can be seen, 97% of the time corresponds to SA. This situation justifies the fact that only the parallelization of the SA algorithm has been undertaken. The next section analyses the different possibilities to parallelize SA that have been reported in the literature to put in context the method here proposed to parallelize SA

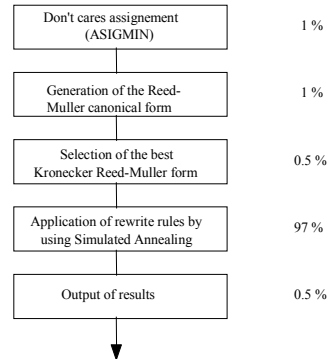


Fig. 1. Flowchart of RRMIN2.

3 Parallelization of Simulated Annealing

As SA provides good results in many combinatorial optimization problems, and as the convergence time is usually high, there have been many attempts to speed up its execution. They can be classified into two major groups. On the one hand we have the procedures that try to optimize the sequential SA algorithm; these either reduce the possibility of rejected transitions in order to decrease the number of iterations to reach

the final solution, or modify the usual annealing schedule, or use specific-purpose architectures [12]. Nevertheless, these procedures reduce the execution time by a factor of two at best.

The second group includes those studies that aim to make use of parallel computers. However, the parallelization of the SA algorithm presents some difficulties, due to its intrinsic sequential characteristics, i.e. each state depends on the previous one, and it is difficult to divide the work to be done at each state among several processors. The studies presented up to now correspond to two main approaches, single-trial parallelism and multiple-trial parallelism.

In single-trial parallelism all the processors in the parallel computer implement a single path in a search tree defined by the possible state transitions that are accepted or not according to the result provided by the evaluation of the cost function. The following are examples of this kind of procedure.

Roussel-Dreyfus Parallel Algorithm [13]. This is a problem-independent parallel SA algorithm that does not change the convergence conditions of the sequential SA algorithm. It identifies two different temperature ranges, determined by the so called acceptance rate $X(T)$. At high temperatures ($X(T) > 1/K$) K processors are used, each processor generates and evaluates a new solution and one of the accepted solutions is randomly selected and sent to all the processors. At low temperatures ($X(T) < 1/K$), it is unlikely that more than one of the K transitions that have been simultaneously tried will be accepted. This solution is sent to other processors. In this situation, a speedup of about K should be obtained. Nevertheless, this algorithm presents two main drawbacks. At high temperatures, a great volume of communications is required, and at low temperatures there is a tendency to select the solutions requiring the least time to be generated.

Speculative Parallel Simulated Annealing. Another strategy to parallelize the SA applies speculative techniques such as Binary Speculative trees (BSC) [14], and Generalized Speculative Computation (GSC) [15], by using generalized N -ary trees. These procedures make a prediction about the future states in the annealing process, thus allowing speedups without a high amount of communications. The common drawback of these procedures appears at high temperatures, when the number of accepted transitions is high and almost all the possible transitions are taken.

Multiple Markov Chains [16]. A further alternative to parallelize SA uses multiple-trial parallelism. This implements a different search path in each processor (using a different seed for each random generation). As the search implemented by each processor can be described by a different Markov chain, this alternative is also called Multiple Markov Chains (MMC). In this kind of procedure, each processor independently applies a transformation to its present solution, evaluates the new solution, and decides whether to accept it or not. In the end, the best solution among those obtained by the processors is taken as the final solution. To reach a suitable speedup, the trials evaluated at each temperature by each processor are reduced by a given factor with respect to the sequential algorithm. The problem with this MMC method is that the conditions under which the convergence of SA is proved are modified, and so the convergence to optimal solutions in the parallel algorithm with MMC, and the results of the sequential SA, are not guaranteed.

Taking into account the pros and cons of each type of parallel procedure, we decided to develop a procedure according to a periodic exchange scheme, based on:

- a) As networks of workstations are going to be used as platforms for the parallel procedure developed here, the reduction of the required volume of communications is an important issue. The MMC schemes require lower levels of communications.
- b) The convergence conditions are not changed, as the Markov chain lengths are the same as in the sequential process. Moreover, if the number of iterations that each processor independently implements is sufficiently high, the convergence is assured for the process implemented in each processor.

4 The Parallel Procedure PRRMIN

A parallel minimization procedure should address two main goals: (a) Speed up the function minimization, and/or (b) Obtain solutions with a better performance in a time similar to that required by a sequential procedure.

Additionally, the volume of communications between processors should be kept as low as possible because the platform to be used in the execution of the parallel procedure is a network of workstations. The AND-EXOR minimization is a problem with a high degree of complexity in which at any state there are a lot of possible transitions. Thus, as the search space presents many branches, methods such as divide-and-conquer, branch-and-bound, or search α - β [17,18] are not very effective because they imply defining a partition in the search space. Moreover, such a partition requires the distribution of different parts of the search space among the processors, thus generating a high amount of communications. To reach goals (a) and (b), the SA process on which the minimization procedure RRRMIN2 is based has been parallelized by using multiple Markov chains with a periodic information exchange scheme. The parallel procedure obtained is called PRRMIN and Figure 2 provides a graphical description about its functioning.

In PRRMIN, all the processors of the machine start from the same initial solution although a different seed is used in the random number generator of each processor (then, each processor explores a different search space). Thus, it is possible to obtain two advantages with respect to the sequential procedure:

- a) Increase the size of search space explored for a time similar to that required by the sequential procedure. In this case, the parallelism is used to obtain better solutions instead of reducing the computing time.
- b) The computing time required by the processors can be reduced whilst the size of the search space explored by all the processors remains similar to that explored when using the sequential algorithm. Thus, solutions with a quality similar to that obtained by the sequential algorithm are obtained by the parallel one but requiring less time.

To reach parallel procedures with an adequate performance, with regard either to solution quality or to speedup, the value of parameter r in Figure 2 should be set to a suitable value. In this way, the value of r would be: (a) Sufficiently high to allow a low volume of communications among processors, making it possible to obtain enough speedup, and (b) Sufficiently small to allow the convergence to solutions with

similar qualities, or even better qualities when the parallel algorithm runs for the same amount of processing time as the sequential one.

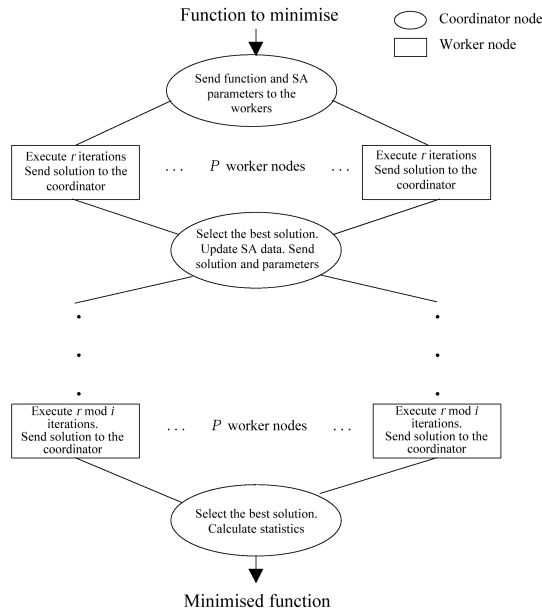


Fig. 2. PRRMIN procedure diagram.

5 Parameter Tuning and Performance Evaluation of PRRMIN

As described in Section 4, PRRMIN implements a SA process which is parallelized using a periodic information exchange among the processors, each implementing a different Markov chain. In this way, different processors apply the SA and generate different paths in the search space. After r iterations, the processors compare the best solution found by each, and proceed in the annealing process by using the best solution. Again, each processor follows a different path in the search space.

Tuning of Parameter r . As indicated at the end of Section 4, the behaviour of the parallel procedure depends, in an important way, on the parameter r , introduced in Figure 2. Thus, this parameter was experimentally tuned by using six SUN IPX workstations at 33 MHz, connected by Ethernet at 10 Mbits/s. The parallel procedure was implemented in C, by using PVM [7].

Figure 3 shows the results of this tuning process. It provides the average number of product terms (MP) against parameter r when the program is applied to series of randomly selected functions with $n=6, 8$, and 10 variables, and 4 processors are used. The number of functions in each series (between 100 and 1000) is such as to allow a maximum error of 5% in the number of product terms with 95% confidence to be obtained. The curves presented in this figure have a minimum for $r=200$, and the same

happens for $p=2$ and $p=6$ processors. Thus, it is possible to conclude that the optimum value for r is independent of the number of variables and processors.

Once the optimal value for r has been obtained, it has been used to evaluate the performances of PRRMIN with respect to the speedup achieved and the quality of solution found.

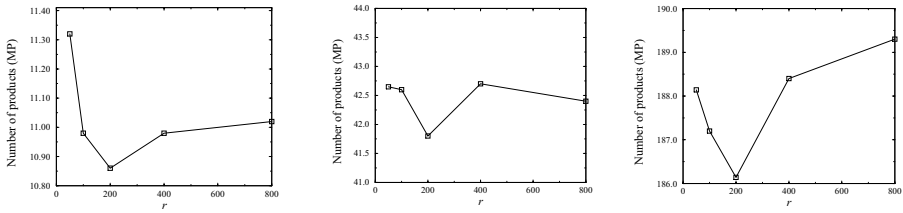


Fig. 3. Number of products vs r for $n=6, 8$ and 10 variables.

Speedup Obtained. To evaluate the speedup obtained by PRRMIN with respect to its sequential version, RRMIN2, some values of TRR (Term Reduction Rate, final number of products divided by initial number of products [18]) are set, and the times to reach them with 1, 2, 4, and 6 processors are measured. As an example, Figure 4 shows the results for random series of functions with $n=6$ variables and $\text{TRR}=0.5$. The speedup obtained for 2 processors is 1.64; for 4 processors, 3.44; and for 6 processors, 4.54. The other curve in Figure 4 provides the speedup for $\text{TRR}=0.4$ with functions of $n=10$ variables. As can be seen, the speedup obtained decreases as the number of variables increases, mainly because the communication requirements also increase very quickly with the number of variables (the number of values to be sent across the network grows as $3n$).

Figure 5 shows the efficiency obtained by PRRMIN as a function of the number of variables and different numbers of processors. The efficiencies for 2 and 4 processors are similar, whereas for 6 processors efficiency is slightly lower.

Nevertheless, the efficiencies are always higher than 65%, and sometimes values close to 85% are reached. These are quite good results if it is taken into account that RRMIN2 is basically a SA procedure, which is an inherently sequential procedure, the fact that AND-EXOR minimization does not allow the partition of the search space, and that communications across the network of workstations are quite slow.

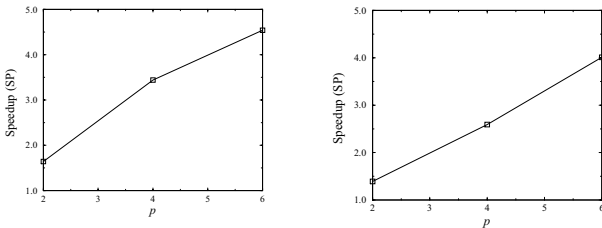


Fig. 4. Speedup vs p for $n=6$ and 10 variables.

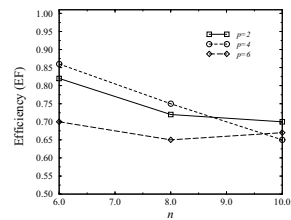


Fig. 5. Efficiency.

Quality of the Solution Obtained. The other alternative to take advantage of the parallel processing implemented by PRRMIN is to improve the quality of the solution found in a fixed amount of time. Thus, an analysis has been made of the improvement

in the solution with respect to an increase in the number of processors (while the computing time is kept constant). Figure 6 shows for $n=6, 8$ and 10 variables, the average number of product terms (MP) against the number of processors. Improvements of up to 10% are obtained in the case of using 6 processors.

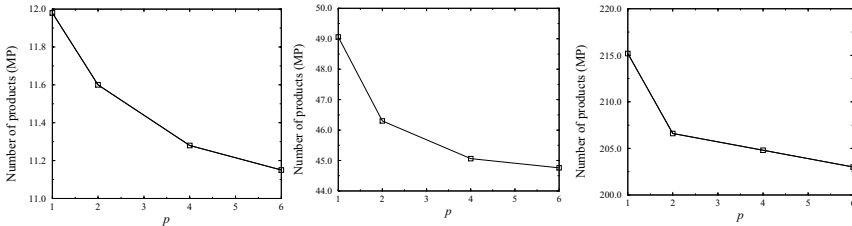


Fig. 6. Improvement for $n=6, 8$ and 10 variables.

6 Conclusions

The AND-EXOR logic allows designs which are simpler, faster, and easier to test than those obtained with AND-OR logic whenever technologies such as dynamic CMOS, FPGAs, etc. are used. This paper presents a parallel procedure to minimize logic functions implemented with AND-EXOR primitives. It provides good results and although it can be run in any multiprocessor platform, it is suited to be executed on networks of workstations.

The procedure, called PRRMIN, is based on the sequential procedure RRMIN2, described in [4]. The parallelization of RRMIN2 is achieved by using multiple Markov chains to distribute the SA process that constitutes the core of RRMIN2. As the processors only communicate after some periods which comprise several iterations, the parallel procedure obtained presents low communication requirements, and is able to reach sufficiently good efficiencies when implemented in networks of workstations. The experiments performed by using 6 SUN IPX workstations connected by an Ethernet network at 10 Mb/s, provide speedups of 4.54 (an efficiency close to 80%), and an improvement of about 10% when the computing time is fixed at the same as the corresponding sequential algorithm RRMIN2.

Thus, a SA with multiple Markov chains and periodic information exchange, appropriately tuned as shown in Figure 3, is able to provide good levels of efficiency when applied to parallelize the sequential procedure RRMIN2. Moreover, the low volume of communications required makes it suitable to be executed in a network of workstations, without causing network overloads that could affect other tasks.

Acknowledgments: This paper has been partially supported by projects PB96-1397 (Dirección General de Enseñanza Superior, Spain) and TIC97-1149 (CICYT, Spain).

References

1. Sasao, T.: Representation of logic functions using EXOR operators. Proc. IFIP WG 10.5 (Reed-Muller '95). Makuhari, Chiba (Japan), (1995), 11-20.
2. Drechsler, R., and Becker, B.: Sympathy: Fast Exact Minimization of Fixed Polarity Reed-Muller Expression for Symmetric Functions. IEEE Trans. on Computer-Aided Design, 16, 1 (1997), 1-5.
3. Garey, M.R. and Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman and Co. San Francisco. (1979).
4. Parrilla, L., Ortega, J., and Lloris, A.: Non-Deterministic AND-EXOR Minimization by Using Rewrite Rules and Simulated Annealing. IEE Proceedings Computer and Digital Techniques. 146, 1, (1999), 1-8.
5. Brand, D., and Sasao, T.: Minimization of AND-EXOR Expressions Using Rewrite Rules. IEEE Trans. on Computers, 42, 5 (1993), 568-576.
6. Parrilla, L., Ortega, J., and Lloris, A.: Using simulated annealing in the minimisation of AND-EXOR functions. Electronic Letters, 30, 22 (1994), 1838-1839.
7. Geist, A., Beguelin, A., Pongarra, J., Jiang, W., Manchek, R., and Sunderam, V.: PVM: Parallel Virtual Machine. The MIT Press, (1994).
8. Aarts, E., and Korts, J.: Simulated Annealing and Boltzmann machines: A stochastic approach to combinatorial optimization and neural computing. Wiley & Sons, (1990).
9. Banerjee, P., Jones, M.H., and Sargent, J.S.: Parallel simulated annealing algorithms for standard cell placement on hypercube multiprocessors IEEE Trans. Parallel and Distributed Systems, 1, (1990), 91-106.
10. Distanto, F., Piuri, V.: Optimum behavioral test procedure for VLSI devices: a simulated annealing approach Proc. IEEE Int. Conf. on Computer Design, Port Chester, (1986), 31-35.
11. Leong, H.W.: A new algorithm for gate matrix layout. Proc. IEEE Int. Conf. on Computer-Aided Design, Santa Clara, (1986), 316-319.
12. Abramson, D.: A very high speed architecture for Simulated Annealing IEEE Computer, 25, 5 (1992), 27-36.
13. Roussel-Ragot, P., and Dreyfus, G.: A Problem Independent Parallel Implementation of Simulated Annealing: Models and Experiments IEEE Trans. on Computer-Aided Design, 9, 8 (1990), 827-835.
14. Witte, E.E., Chamberlain, R.D., and Franklin, M.A.: Parallel simulated annealing using speculative computation. IEEE Trans. on Parallel and Distributed Systems, 2, 4 (April 1991), 483-494.
15. Shon, A.: Parallel N-ary Speculative Computation of Simulated Annealing. IEEE Trans. on Parallel and Distributed Systems, 6, 10 (1995), 997-1005.
16. Lee, S.-Y. and Lee, K.G.: Synchronous and Asynchronous Parallel Simulated Annealing with Multiple Markov Chains. IEEE Trans. on PDS, 7, 10 (1996), 993-1007.
17. Horowitz, E. and Zorat, A.: Divide and Conquer for Parallel Processing IEEE Trans. on Computers, C-32, 6 (1983), 582-585.
18. Yang, M.K., and Das, C.R.: Evaluation of Parallel Branch-and-Bound Algorithm on a Class of Multiprocessors IEEE Trans. on Parallel and Distributed Systems, 4, 1 (1994), 74-86.
19. Lloris, A., Gómez, J.F., and Román, R.: Entropic minimization of multiple-valued functions. In IEEE Proc. of the Int. Symp. on Multiple Valued Logic, (1993), 24-28.

Author Index

- Albers, B., 418
Alexandrov, V., 283
Almeida, F., 27
Andrés, B. de, 291
Arpaci-Dusseau, A.C., 215
- Badía, J. M., 372
Banaś, K., 349
Baraglia, R., 364
Bassomo, P., 396
Bertozzi, M., 199, 426
Beyls, K., 173
Blaheta, R., 299
Błaszczuk, A., 493
Borkowski, J., 157
Boselli, F., 199
Bosschere, K. De, 141
Boudet, V., 333
Brignole, N. B., 533
Bubak, M., 59, 67
Bukowski, M., 83
- Calderón, A., 207
Carretero, J., 207
Chassin de Kergommeaux, J., 141
Chaussumier, F., 485
Chergui, J., 341
Čiegis, R., 275
Claver, J. M., 388
Clematis, A., 101
Connolly, R. W., 181
Conte, G., 199, 410
Corbacho-Lozano, J., 477
Corbel, A., 396
Cownie, J., 51
Culler, D.E., 215
Czarnul, P., 509
- Desprez, F., 485
D'Hollander, E., 173
Doallo, R., 133
Dobrucký, M., 450
- Domingo-Pascual, J., 477
Domokos, G., 267
- Eberl, M., 493
Eickermann, T., 3
Espinosa, A., 91
Esteban, S., 291
Evripidou, P., 249
- Fava, A., 410, 426
Fava, E., 410
Fava, M., 426
Ferenc, D., 257
Fernández, F., 241
Ferrini, R., 364
Frugoli, G., 410
Funika, W., 67
- Gallud, J. A., 442
García-Consuegra, J., 442
García, F., 207
García, I., 380
García, J. M., 442
Garzón, E. M., 380
Gažák, M., 517
Gianuzzi, V., 101
Godlevsky, A., 517
Gómez, J. A., 241
González, J.A., 27, 189
Gropp, W., 11, 51
Grund, H., 3
- Halada, L., 450
Hempel, R., 109
Henrichs, J., 3
Hernández, V., 388
Hidalgo, J. I., 291
Hiraki, K., 19
Hluchý, L., 450, 517
Huse, L. P., 469
- Ishihara, S., 461

- Iskra, K., 67
- Jakl, O., 299
Jonker, P., 418
- Karaivanova, V., 283
Karl, W., 493
Kitowski, J., 349
Kranzlmüller, D., 43
Krawczyk, H., 509
Kutil, R., 149
Kuzora, P., 83
- Laforenza, D., 364
Laganà, A., 364
Lanchares, J., 291
Laohawee, P., 325
Larsen, M., 356
Lepe-Aldama, O., 477
León, C., 27, 189
Llorente, I. M., 307
Lloris, A., 541
Lopes, L., 525
Luque, E., 91
Lusk, E., 11
Łuszczek, P., 59
- MacFarlane, A., 317
Madsen, P., 356
Margalef, T., 91
Maruszewski, R., 67
Matsumoto, T., 19
Matuszek, M. R., 499
Mazurkiewicz, A., 499
McCann, J. A., 317
Mierendorff, H., 75
Migliardi, M., 117
Mollar, M., 388
Morales, D.G., 27
Morimoto, K., 19
Morrison, J. P., 181
Mourão, F. E., 231
- Nabrzyski, J., 257
Neyman, M., 83
- Nicolescu, C., 418
- Ortega, J., 541
Papagapiou, A., 249
Parcerisa, F., 91
Parrilla, L., 541
Paulino, H., 525
Płazek, J., 349
Prieto, M., 291, 307
Prylli, L., 223, 485
- Rabenseifner, R., 35
Rastello, F., 333
Reggiani, M., 199
Reussner, R., 43
Ritzdorf, H., 109
Rivera, D., 291
Robert, Y., 333
Robertson, S. E., 317
Roda, J.L., 27, 189
Rodríguez, C., 27, 189
Ronsse, M., 141
Rungsawang, A., 325
- Šablinskas, R., 275
Sakho, I., 396
Samaras, G., 249
Sánchez, J.M., 241
Sande, F. de, 27, 189
Santiago, R., 307
Schaubschläger, C., 43
Schuele, J., 404
Schwamborn, H., 75
Silva, F., 525
Silva, J. G., 125, 231
Silva, P., 125
Solé-Pareta, J., 477
Stankovic, N., 165
Starý, J., 299
Stroiński, M., 257
Sunderam, V., 117
Szeberényi, I., 267
- Takahara, A., 461
Tangpong, A., 325

Tani, S., 461
Tirado, F., 291, 307
Tomassini, M., 241
Tourancheau, B., 223
Tourinho, J., 133
Traff, J.L., 109
Tran, V. D., 450
Trinitis, C., 493

Uhl, A., 149
Uminski, P. W., 499

Vazquez, G. E., 533

Vidal, A. M., 372

Wasniewski, J., 275
Westrelin, R., 223
Wierzejewski, P., 257
Wismüller, R., 67
Wong, F. C., 215
Wu, P.-Y., 434

Yu, Y., 173

Zimmermann, F., 109